NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

SIMULATING CLOUDS WITH PROCEDURAL TEXTURING TECHNIQUES USING THE GPU

by

Georgios E. Tarantilis

September 2004

Thesis Advisor:	Rudy Darken
Co-Advisor:	Joe Sullivan
Second Reader:	Erik Johnson

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2004	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Simulating Clouds with Procedural Texturing Techniques Using the GPU			**5. FUNDING NUMBERS**
6. AUTHOR(S) Georgios E. Tarantilis			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**
13. ABSTRACT (maximum 200 words) Many 3D training simulations employ static, and to some extent, simplistic natural phenomena representation that often leaves much to be desired. Taking advantage of the latest advancements in computer graphics hardware allows modeling dynamic natural phenomena such as clouds. Specifically, utilizing procedural techniques and high-level shading languages, it is possible to produce considerably more realistic simulations. This thesis designed and implemented a visual simulation component, which renders convincing clouds using procedural noise-based texture mapping techniques. Both traditional rendering and shader-enabled rendering supported by the OpenGL Shading Language are utilized. This component has been included in the Delta3d simulation engine and is used to create convincing clouds in outdoor simulations while the performance penalty imposed is considered acceptable. Custom tools have been developed for easy noise texture parameterization and cross-platform compatibility has been demonstrated.			
14. SUBJECT TERMS 3D Training Simulations, Clouds, Convincing Clouds, Rendering, Shader-Enable Rendering, OpenGL Shading Language, Delta3d Simulation Engine, Texture Parameterization, Cross-Platform Compatibility			**15. NUMBER OF PAGES** 69
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

SIMULATING CLOUDS WITH PROCEDURAL TEXTURING TECHNIQUES USING THE GPU

Georgios E. Tarantilis
Lieutenant, Hellenic Navy
B.S., Hellenic Naval Academy, 1993

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MODELING VIRTUAL ENVIRONMENTS AND SIMULATIONS (MOVES)

from the

NAVAL POSTGRADUATE SCHOOL
September 2004

Author: Georgios E. Tarantilis

Approved by: Rudy Darken
Thesis Advisor

Joe Sullivan
Thesis Co-Advisor

Erik Johnson
Second Reader

Rudy Darken
Chairman, MOVES Academic Committee

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Many 3D training simulations employ static, and to some extent, simplistic natural phenomena representation that often leaves much to be desired. Taking advantage of the latest advancements in computer graphics hardware allows modeling dynamic natural phenomena such as clouds. Specifically, utilizing procedural techniques and high-level shading languages, it is possible to produce considerably more realistic simulations. This thesis designed and implemented a visual simulation component, which renders convincing clouds using procedural noise-based texture mapping techniques. Both traditional rendering and shader-enabled rendering supported by the OpenGL Shading Language are utilized. This component has been included in the Delta3d simulation engine and is used to create convincing clouds in outdoor simulations while the performance penalty imposed is considered acceptable. Custom tools have been developed for easy noise texture parameterization and cross-platform compatibility has been demonstrated.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM STATEMENT

The technological achievements in the computer hardware field during recent years have been astounding. Visual simulations are taking advantage of new features and capabilities of graphics hardware, thus taking the degree of realism and visual fidelity to higher levels. One of the fields still experiencing major research is the simulation of natural phenomena. These are essential in presenting outdoor scenes where the user or trainee must be fully immersed in the environment. Additionally, many training applications must provide the trainee with cues such as wind direction and shadows in order to prevent negative transfer of training. Albeit, nature is unpredictable and does not follow a definite set of rules that can be modeled easily with computer algorithms and logic. That being the case, developing a precise visual model of a single natural phenomenon is feasible under a large number of assumptions, but generally, the computation and rendering process will consume so many CPU (or GPU) cycles that it cannot be executed at interactive rates.

For this reason, many visual simulations of outdoor scenes use procedural texturing techniques to catch the irregularities and randomness found in nature. Requirements for creating textures programmatically are parameterization and the property of being repeatable without any noticeable seams when these textures are tiled. These procedural generated textures can be used by either traditional rendering applications or applications using the new programmable features of graphics hardware.

The research areas of this thesis are

- The creation of procedural 2D and 3D noise-based textures
- Cloud visual simulation through the use of regular texture mapping technique and a GPU shader-enabled updateable texture technique
- Integrating this cloud simulation into an open source 3D visual simulation and game engine

B. RESEARCH QUESTIONS

Clouds are an important element of the visual simulation of any outdoor scene and their inclusion augments the realism and visual appeal of the simulation. Clouds are rarely examined closely in life. They are just part of the environment and expected to be there. The goal is to integrate convincing procedural clouds in an outdoor simulation.

The research questions that need to be answered are the following:

- How can noise-based procedural textures be created? Which pixel format will be used? In which file format will it be stored?
- How can cloud-like textures be created out of plain noise-based textures?
- How it is possible for the process to be parameterized in order to easily create many different textures?
- Is there a minimum requirement for graphics cards in order to support the shader-enabled technique?
- Which shading language should be used to implement the shaders?
- What are the restrictions in the simulation?
- How is it to be integrated with Delta3D simulation engine?
- Is it possible for the simulation to be executed in many platforms?
- What is the performance penalty introduced? Is the framerate drop significant?
- When is it appropriate to use this simulation?

II. BACKGROUND

A. GRAPHICS HARDWARE

1. Evolution

Computer graphics hardware has evolved dramatically during the past five years. It has advanced from the graphics accelerators of PCs and workstations, which simply were giving a performance boost, to the Graphics Processor Unit (GPU), which is capable of supporting complex, user-programmable shading programs with high performance.

The driving forces behind this progress are undoubtedly the vast amount of computation needed to simulate the world along with the human desire to be visually stimulated and entertained. Aided by the semiconductor industry, which has committed itself to doubling the number of transistors that fit on a microchip every 18 months (Moore's Law), GPUs have been highly specialized and not only has performance increased greatly, but the quality of computation and the flexibility of graphics programming have also steadily improved.

2. Programmable Hardware

Until recently, the limited functionality of the fixed graphics pipeline restricted developers in what they could create in real-time. This generally resulted in synthetic-looking real-time graphics. However, the world is comprised of very complex materials and lighting effects, so real-time graphics hardware had to support shading models other than the fixed pipeline provided. This was possible by making certain stages in the sequence of the pipeline programmable and by providing support from the two graphics rendering APIs, DirectX and OpenGL, which exposed these new hardware capabilities to the end-user. The developer now can program the vertex and fragment processor within the GPU bypassing the fixed pipeline and create realistic graphics using consumer video cards. For visual simulation applications specifically, real-time programmable shading enables superior visual realism.

3. Future Trends

The latest graphics cards augment the programmability of the hardware adding features such as the ability to read texture memory in vertex programs and branching in fragment programs. Future graphics cards will likely further expand hardware programming capabilities along with increased performance and flexibility. Support for algorithms such as noise evaluation functions and global illumination have been announced for the graphics hardware.

Additionally, NVIDIA has announced (July 2004) its SLI (Scalable Link Interface) multi-GPU architecture. This technology promises to take advantage of the increased bandwidth of the PCI Express bus architecture and allow multiple GPUs to work efficiently in parallel in a single system.

B. HIGH-LEVEL SHADING LANGUAGES

The first high-level shading language to become the industry standard for offline rendering systems was PIXAR's RenderMan Shading Language in 1988. Such languages are using the CPU for rendering and are non-interactive but they greatly influence the invention of real-time high-level shading languages.

Most graphics hardware and APIs support programming the GPU with low-level programming interfaces, usually at the assembly language level. Unfortunately, it is rather difficult and often unproductive to program in assembly language. Thus, it was necessary to create high-level shading languages. These high-level languages would offer advantages such as hardware abstraction and faster compiler-optimized output code.

Real-time high-level shading languages have leveraged the accumulated knowledge and shader techniques of offline languages and further provided true interactivity (change of viewpoint) and performance. Also, they have potentially better performance than assembly programming because shaders are optimized by the compiler.

Currently, there are three main rivals in the field of shading languages:

- NVIDIA's Cg
- Microsoft's HLSL
- OpenGL Shading Language (GLSL)

Cg and HLSL have essentially the same syntax and capabilities because NVIDIA and Microsoft collaborated in their creation. The main goal is to compare these two with GLSL and decide which will have greater developer acceptance, performance and integration in applications and graphics APIs. This was a turning point for this thesis because these languages evolve radically and it is difficult to predict their future.

A key difference between GLSL and the other languages is that GLSL is part of the OpenGL 1.5 API specification and does not need any other translation between the application, the shader source code and the API calls. Of course, if one uses the other competing API (DirectX), this is not a problem. In general, all high-level languages compile or translate C-like source to machine code to be executed in the appropriate API.

C. PROCEDURAL TEXTURING

1. Introduction

There have been two recent important developments for real-time procedural texturing techniques: increased CPU power and programmable graphics hardware (GPUs), which are available on commodity PCs. This has allowed graphics developers to create interactive complex procedural effects.

One of the most important features of procedural texturing is abstraction. In a procedural approach, rather than explicitly specifying the texture that will be applied to the scene models, they are abstracted into a function or an algorithm (i.e., a procedure) and evaluate this procedure when needed. Other advantages of procedural texturing are:

- *Parametric control* makes it possible to create variations of the same theme easily
- *Variable resolution* offers "infinite" detail, limited only by precision
- Procedural textures can cover arbitrarily large areas, with *no repeating*
- Solid texturing – not limited by texture mapping coordinates because evaluated textures adapt to arbitrary geometry
- It is only code - it requires minimal memory storage compared to textures

The most significant drawback of procedural texturing, at least currently, is that it requires many computation cycles, so that it is difficult to compute textures, especially

large 3D textures, in the GPU at interactive rates. Therefore, the textures are usually precomputed on the CPU and passed onto the GPU. Hopefully, this disadvantage will vanish when newer and faster GPUs are developed.

One other difficulty presented to the developer of procedural textures is that this is a highly iterative process, and writing and debugging such code is not always intuitive. Given too little parameterization, it is not possible to create what was envisioned. Given too much, and it is possible to become mired in thousands of iterations.

2. Noise

What makes natural phenomena and objects unique in their appearance is their apparent randomness. Various noise functions are used to simulate that and other phenomena in procedural texturing. They replace or modulate repeating textures with procedural ones adding controlled randomness to them. The first implementation of a noise function for procedural texturing was done by Ken Perlin [3], for which he also received an Academy award. Since then, this function has provided inspiration, and numerous variations of value, lattice, gradient, and fractals have been presented. Figure 1 presents two examples of 1D noise functions. Noise functions have been widely used in the film industry, commercials, and computer graphics for many years.

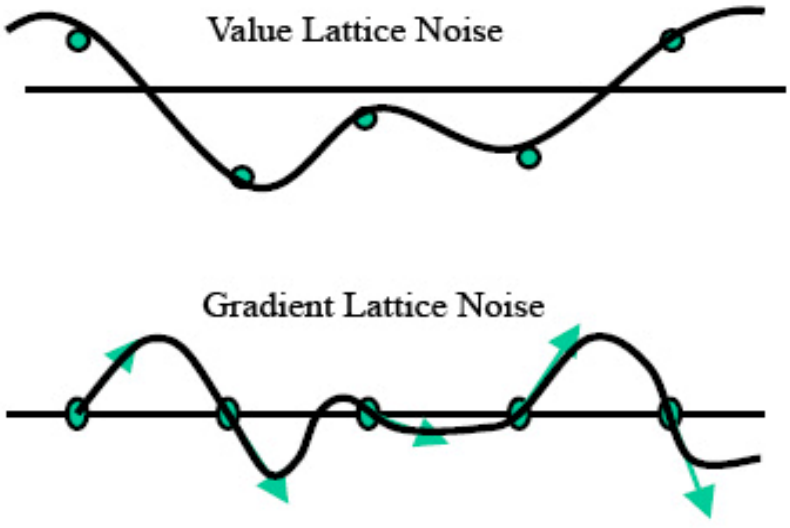

Figure 1. 1D Noise Functions

The ideal noise function characteristics are:

- Repeatable pseudorandom values
- Specific range (typically [-1,1] or [0,1])
- No repeating patterns
- Invariance under rotation and translation

3. Solid Noise Textures

Building a 3D noise texture, also called solid texture or volume texture, requires generating the 2D texture slices that will be stacked together. OpenGL 1.3 and later drivers, natively support 3D textures, while other earlier drivers optionally support them through extensions. To ensure 3D texture will be continuous in space these 2D textures slices must be tileable in both the x and y axes.

Another issue is that 3D textures quickly become very large with increased resolution. In other words, a 128x128x128 RGBA texture occupies 128^3x4 bytes = 8Mb of memory, which eventually will be loaded into the graphics card. A 256x256x256 RGBA texture occupies 256 Mb, which most current cards simply do not have.

Thus, it is necessary to select a smaller resolution, and at the same time, take steps to ensure that the visual appeal of the texture is acceptable. Filtering (tri-linear, cubic filtering), texture compression and noise function optimizations are some tools that exist for this function, undoubtedly with some trade-offs, as far as performance is concerned.

4. Natural Phenomena

Procedural texturing can mimic statistical properties of natural textures and can be used in many applications for the simulation of natural phenomena and effects such as clouds, gases, smoke, water, and terrain. Many of these simulations follow purely empirical approaches that make everything look convincing, but have nothing to do with physics and material properties. The world is far too computationally complex to model every aspect with fine detail. Nevertheless, there have been some physical-based simulations ([2][10]), which use simplified mathematical models that are both computationally tractable and aesthetically pleasing.

D. SCENE GRAPHS

1. Introduction

A scene graph is a hierarchically arranged data structure that encapsulates and describes the 3D world of a simulation: 3D models, lights, cameras, and actions. It is a directional, acyclic graph (DAG) or more commonly, a tree. Its structure determines the order of operation of its data and different node types provide mechanisms for grouping, animation, level of detail, and other concepts that are applied when the scene graph is traversed.

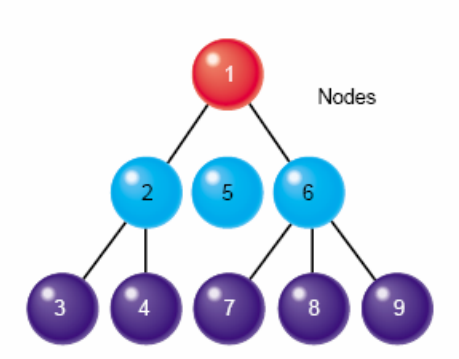

Figure 2. Scene Graph Structure

A primary role of a scene graph is to improve performance through culling, state sorting and various other methods, which reduce the load on the graphics rendering engine, allowing complex scenes to be rendered faster.

2. OpenSceneGraph

OpenSceneGraph (OSG) is a multi-platform open source graphics toolkit for the development of graphics applications such as flight simulators, games, virtual environments or scientific visualization. Written entirely in Standard C++ and OpenGL, it makes use of the Standard Template Library (STL) and Design Patterns, and leverages the open source development model to provide a library that keeps pace with the graphics hardware evolution.

The Delta3D simulation engine being developed in the MOVES Institute of the Naval Postgraduate School has chosen OSG as the framework upon which to be built and extend its functionality. Delta3D's goal is to be used for simulations, military training

applications, games, or other graphical applications. It provides a high-level API while still allowing the end-user optional, low-level functionality. Additionally, it is based completely on open source software so it is both flexible and expandable in response to future needs.

THIS PAGE INTENTIONALLY LEFT BLANK

III. CLOUD SIMULATION DESIGN

A. INTRODUCTION

1. Purpose

The purpose of this thesis was to research real-time natural phenomena simulation, more specifically cloud simulation, in virtual environments applications, and possible uses of GPU programming techniques. The developed libraries were included in the simulation engine Delta3D (former P51), an ongoing project currently sponsored by N6M (U.S. Navy Modeling and Simulation Management Office).

2. Overview

The initial thought was to divide the research for cloud simulation into three topics using both traditional rendering techniques through the fixed pipeline and programmable graphics hardware:

- Traditional textured-mapped cloud planes (rectangles) with run-time procedural *static* textures above the 3d world
- Dome-shaped (hemispherical structure) cloud surface around the 3d world rendered with direct manipulation of a run-time or pre-computed procedural volume texture in the GPU. Texture is updateable through vertex and fragment shaders
- Volumetric clouds using techniques of implicit surface modeling [2] and volume rendering with programmable graphics hardware

The first two approaches have been designed, tested and developed. The third approach, after conducting some preliminary research on the methods of implicit modeling (blobs, isosurfaces) and volumetric rendering with shaders, was considered unfeasible to be attacked in the timeframe required. Nevertheless, this method should be used in visual simulations when the viewpoint is not consistently near ground level, as with a flight simulator.

The three main classes from this research, *NoiseGenerator, CloudPlane* and *CloudDome*, have been developed as a part of a Delta3D simulation engine, but with minor changes, they can be used from whichever application or simulation uses the OpenSceneGraph framework.

3. Tools

Standard C++ language and Microsoft Visual Studio .NET 2003 as the IDE were used to develop the libraries. The API of choice was OpenGL 1.5, and the graphics library used was OpenSceneGraph. The *OpenGL Shading Language* and ATI's shader IDE *RenderMonkey* 1.5 (with GLSL support) was used as a shading language to develop, preview and tweak the shaders. Additional tools were Microsoft's *DirectX Texture Tool* (part of DirectX 9.0b SDK) for viewing and manipulating 2D and volume DDS files, Adobe's *Photoshop* for image editing, NVIDIA's *DDS plugin* for Photoshop, the custom-made tool *"Make Some Noise"* to augment the process of noise texture generation, and Troll Tech's *Qt*.

B. DESIGN DESCRIPTION

1. Overview

The presentation of the whole hierarchical structure of the *Delta3D* simulation engine is beyond the scope of this thesis, but the classes relevant to this research must be discussed. The three pertinent classes are the *EnvEffect* class, the *Environment* class and the *Weather* class. *EnvEffect* is a base class for any environmental effect class, so *CloudPlane* and *CloudDome* classes inherit from it (Figure 3).

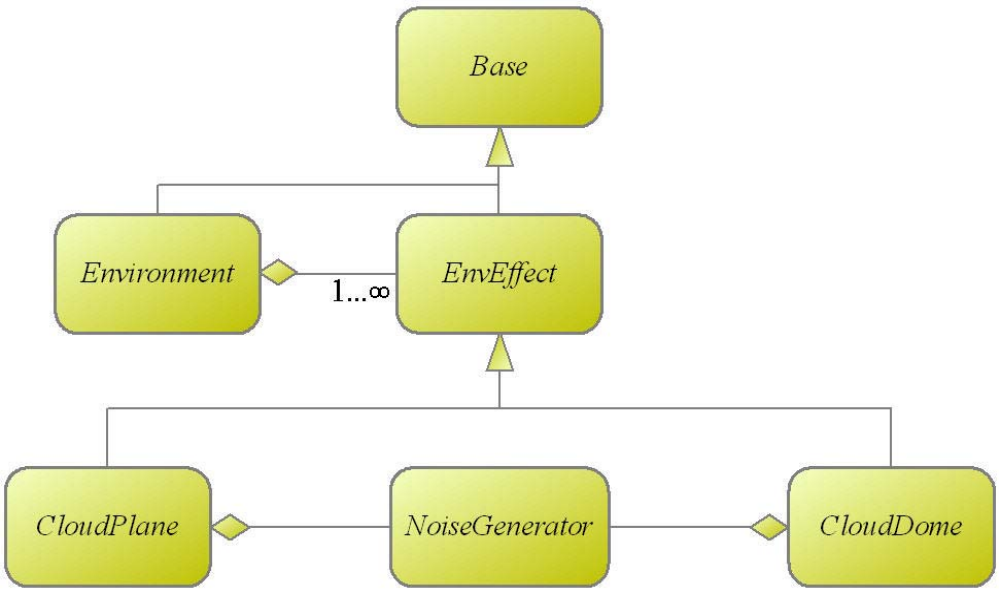

Figure 3. Class Hierarchy

Environment maintains a list of effects and any new effect must be registered to this list in order to be rendered. Multiple instances of *CloudPlane* can be added to the *Environment* to represent layers of clouds, although only one instance of *CloudDome* can exist at a time. Normally, the two classes are used interchangeably but not both at the same time. Developers can embody the required functionality by using either of the two for rendering clouds depending on each application's needs.

The *Environment* class provides a mechanism for the registered environmental effects to be updated at runtime by passing variables such as *skyColor, fogColor,* and *sunAngle* in the *Repaint.* This update happens every second but if an *EnvEffect* needs to be updated for every frame, it is possible to register the *System* class as a Message Sender and obtain the *deltaFrameTime*, which is the time passed from frame to frame.

2. NoiseGenerator Class

The *NoiseGenerator* class is used to create procedural 2D and 3D gradient lattice noise textures for use by the other two classes *CloudPlane* and *CloudDome*. The algorithm of the noise function is based on Ken Perlin's improved Noise [3]. This improved algorithm reduced the grid-oriented artifacts by introducing a small set of fixed gradient directions and replaced the cubic interpolation of *fade* function with a fifth order polynomial to eliminate discontinuities of second and third order derivatives. This implementation was written in Java and had to be ported to C++.

Additional extensions added were the support for 2D noise and the extremely important property of wrapping in every axis that produced tileable 2D and 3D noise. The texture is created by direct manipulation of the image data segment through extensive use of pointers. Also, the three most frequently used functions in the noise generation process (*fade, lerp* and *grad*) were inlined to improve performance.

The user can parameterize the following inputs of the *NoiseGenerator* class constructor:

- *Octaves* of noise
- Initial *Frequency*
- Initial *Amplitude*

- *Persistence*
- Texture dimensions (*Width, Height, Slices=Depth*)

The end user, by tweaking these parameters, can create an infinite number of noise textures. For example, Figure 4 has two sample 2D and one 3D noise textures. The left texture of the two 2D textures has six octaves, initial frequency=6, initial amplitude=0.7 and persistence=0.5, while the right has four octaves, initial frequency=3, initial amplitude=1 and persistence=0.3

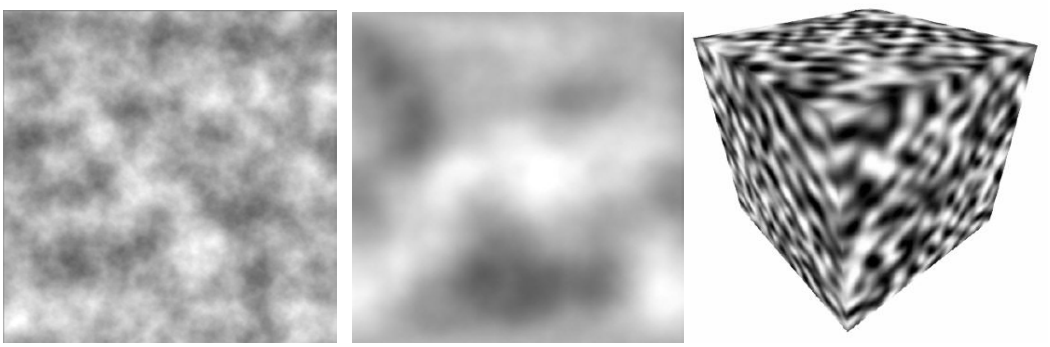

Figure 4. Samples of 2D and 3D Noise Textures

The resulting texture is seamless the same texture is arranged side-by-side in every direction. For example in Figure 5 shows the middle texture tiled four times.

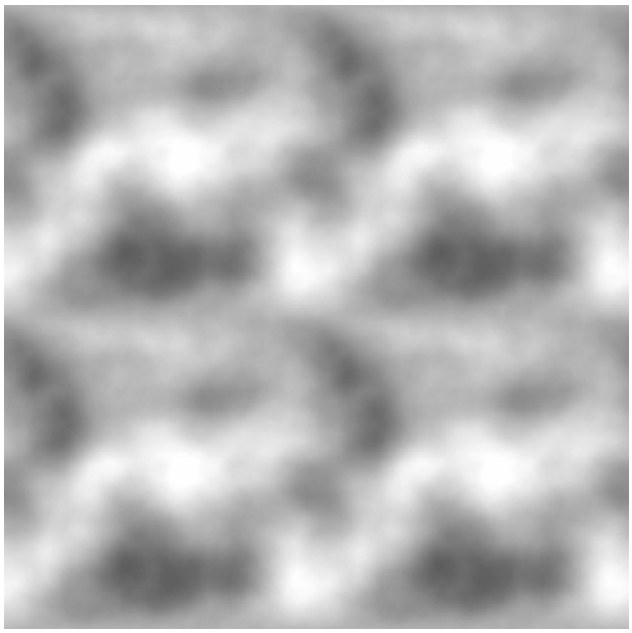

Figure 5. Tiled Pattern

This property was achieved by passing the frequency parameter to the *noise* function and apply the modulo operator (%) to the integer parts of the coordinates x, y and z. This operator provides the remainder of the division of two numbers and has the useful property of wrapping when applied in a sequence of numbers. For example, assume the frequency is 5. When the modulo operator is applied to a series of integer numbers and the frequency, it will yield:

$$1\%5 = \underline{0.2}\ ,\ 2\%5 = 0.4\ ,\ 3\%5 = 0.6\ ,\ 4\%5 = 0.8\ ,\ 5\%5 = 0\ ,\ 6\%5 = \underline{0.2}$$

The random numbers that initialize the permutation array are created with the *reseed* function, which uses *rand* and as seed, the machine time to avoid any similarities in subsequent calls.

The available pixel formats are ALPHA, LUMINANCE, RGB or RGBA. The most efficient formats are ALPHA and LUMINANCE because they use only one byte per pixel. In other words, if a texture is stored in ALPHA format, it requires ¼ of memory and storage space than the same texture stored in RGBA format. Besides, no information is lost because the texture stored in RGBA format is grayscale. Of course, later the texture stored in ALPHA format must be subjected to special treatment because this channel is not normally visible.

Another solution to reduce the size of the textures, especially the volume ones, was to use a compression technique. OpenGL 1.3 and later versions provide native support for texture compression. The compression format is dependent on the implementation of the driver but if the extension *GL_EXT_texture_compression_s3tc* is present in the system, the S3TC_DXT formats could be used. The caveat using one of these compression formats is that they are not *lossless*. After doing some experimentation with compressed noise textures, it was concluded that the visual fidelity of the rendering was unacceptable.

Although the produced textures appear as controlled random noise, they need to be processed to have a cloud-like appearance. Two different techniques were used for the

two cloud classes, respectively, which are described in the appropriate sections. In general, the pixel values of the texture must be clamped to create patches of clouds and then exponentiated to decrease the dynamic range.

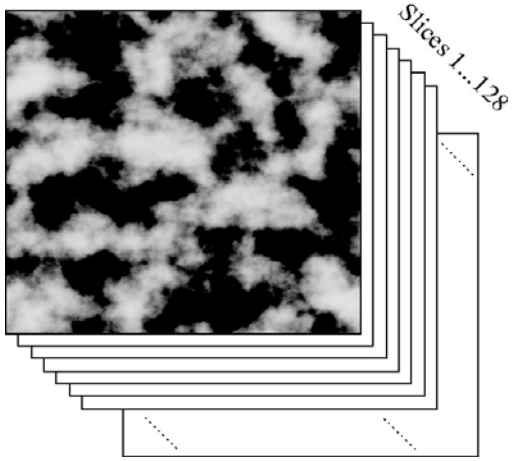

Figure 6. 3D Texture Structure

One feature worth mentioning is how the 3D textures are created. OpenGL version 1.2 and later versions support 3D textures, which essentially are a series of regular 2D textures. The interesting characteristic of 3D textures is that when they are applied to an object with proper texture coordinates, the graphics hardware interpolate between the 2D texture slices. The interpolation method, which is the same with individual 2D textures, controls the final image quality.

3. CloudPlane Class

a. Overview

The *CloudPlane* class is registered with the *Delta3D* framework by inheriting from the *EnvEffect* class. It is used to create layers of clouds above the 3d world of the simulation. This is achieved by creating pseudo-infinite planes (quads with very large dimensions), which are properly tessellated and textured to give the impression of clouds. These planes must be "fixed" above the viewer, which is achieved by inserting their nodes as children of a transform node (*MoveEarthySkyWithEyePointTransform*) that automatically translates them to the viewer point coordinates.

b. Noise Texture

The *NoiseGenerator* class is used to create the required 2D noise texture, which is subsequently processed to generate the cloud-like texture. The four necessary parameters for creating the noise texture (noise octaves, initial frequency, initial amplitude and persistence) are passed to the *NoiseGenerator* class along with the required dimensions of the texture. The dimensions should be equal in each direction and should be a power of 2. The recommended value for width and height is 512x512 since lower values do not produce visually acceptable results and higher values take much more time to be computed. Further required processing of generated noise texture is clamping and exponentiation, which will produce noise textures with cloud-like appearance and is described in detail in the section entitled "Post-Process of Noise Textures".

The texture mipmaps are set to be generated automatically with the function call *mTexture→setUseHardwareMipMapGeneration(true)*. The minification and magni-fication texture filtering is set to LINEAR_MIPMAP_LINEAR in order to obtain the smoothest antialiasing possible. In addition, when associating the color and normal arrays with the geometry, the binding mode for the array of colors is set to BIND_PER_VERTEX and for the normals is set to BIND_OVERALL.

It is necessary to allow the sky to be seen through the cloud layers and this is achieved by using a blending equation with source and destination factors SRC_ALPHA and ONE_MINUS_SRC_ALPHA, respectively. Since the generated texture has only the ALPHA channel available, the blending is automatic. The only restriction that exists is that the layers should be added in a lower-to-upper order because, otherwise, they will not be rendered correctly. This is a limitation of OpenGL transparency management and it is not possible to circumvent it using only the fixed graphics pipeline model.

The initial approach of assigning a single quad as a cloud layer was not successful due to the problem of hard edges near the horizon. Dividing the quad in nine segments and setting appropriate values for the vertices ALPHA values solved this problem. The outer vertices have ALPHA equal to zero, while the four inner vertices

have ALPHA equal to one. Later in the rasterization phase in the pipeline, the intermediate pixels between the outer and inner edges will be assigned the interpolated ALPHA value between zero and one. This ALPHA value combined with the ALPHA value of the applied 2D noise texture will provide the final ALPHA value that will be written in the color frame buffer. Figure 7 shows a representation of the discussed method. The numbers indicate the segment divisions. Figure 8 demonstrates a comparison between a regular four-vertex quad and one that uses the tessellated version. It is apparent how the tessellated quad allows for smooth edges in the horizon.

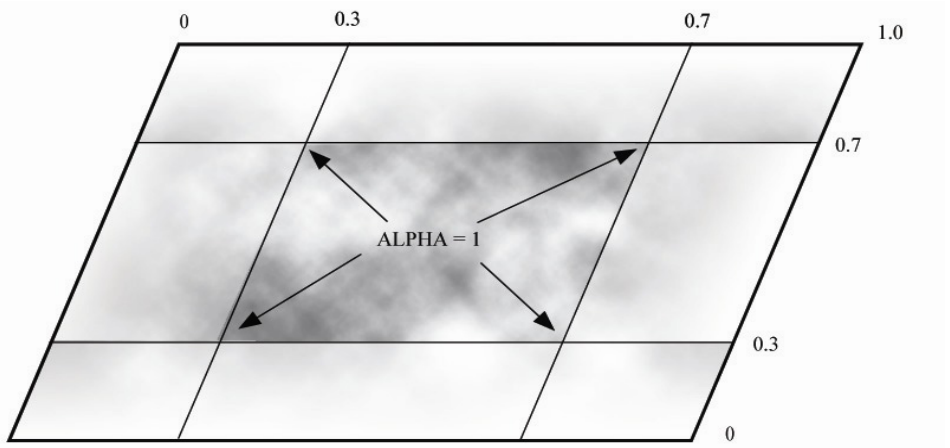

Figure 7. Tessellated CloudPlane Quad

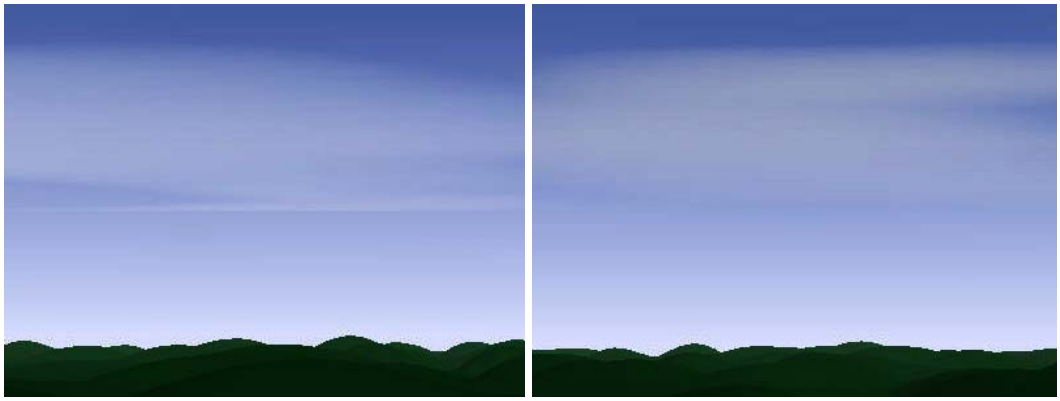

Figure 8. CloudPlanes with Hard Edges and Soft, ALPHA Enabled, Edges

The textures are applied on the quads with texture coordinates that are modified at runtime to give the impression of wind because the clouds appear as though they are moving. This is achieved with the *mWind* parameter (an *osg::Vec2* instance),

that modifies each layer texture coordinates relative to their height from the ground. Every cloud layer has a user-defined height (*mHeight)* above the ground, which impacts the effect that wind has on it. The closer to the ground the layer is, the greater its speed so as to give the impression that what is seen in real life are low-height fast-moving clouds.

c. Update

Two methods were followed to change the various parameters of the cloud layers. The first method implements the virtual function *EnvEffect::Repaint().* This function is required to be implemented by all classes that inherit from *EnvEffect* and is called automatically by the *Environment* class every second. The following *Environment* parameters are updated: *sky_color, fog_color, sun_angle, sun_azimuth* and *visibility*. By using these values, it is straightforward to compute the updated cloud layer color (*mCloudColor*), and along with the *fog_color*, to determine the final color of the inner out outer vertices of the quad. Then, these colors are updated with the call *mPlane→setColorArray(mColors)*.

The second method is to subscribe the *CloudPlane* class to receive messages from the *dtCore::System* class with the *AddSender(System::GetSystem())* call. Any message that *System* sends will be received in the *CloudPlane::OnMessage()* method[1]. In particular, the two messages of interest are "preframe" and "postframe", which contain the *deltaFrameTime* value that denotes the time that passed from one frame of the simulation to the other.

Although the value *deltaFrameTime* was not needed in this class, the second method was used for updating the texture coordinates of the quad because the *Repaint* function of the first method is not called every frame and a "choppy" appearance in the cloud layers appeared.

4. CloudDome Class

a. Overview

The *CloudDome* class is also registered with the *Delta3D* framework by inheriting from the *EnvEffect* class. It renders clouds on a dome-shaped (hemispherical structure) cloud surface around the 3D world by direct manipulation of a run-time or pre-

[1] The mechanism that supports the messaging system relies on the Sig-Slot architecture.

computed procedural volume texture in the GPU. Both pre-computed and run-time volume noise textures are being generated by the *NoiseGenerator* class. The texture is updated through vertex and fragment shaders written in the OpenGL Shading Language at runtime with use of user-defined uniform variables either programmatically or with a GUI applet.

b. Dome

The dome 3D object is constructed by a series of triangle strips with the arrangement to have larger density where the alpha transition occurs in order to be smoother. This is achieved by setting the angles from the bottom up to appropriate values: the strip altitude increments are 0°, 4°, 5° and 5° with corresponding alpha values for the vertices at that levels 0, 0.3, 0.7 and 1.0 (Figure 9). The actual colors of the dome vertices are bright red, green and blue but normally they are not active in the simulation because they are used only for debugging purposes when the shaders are inactive. The number of levels used is seven while the number of segments is twenty.

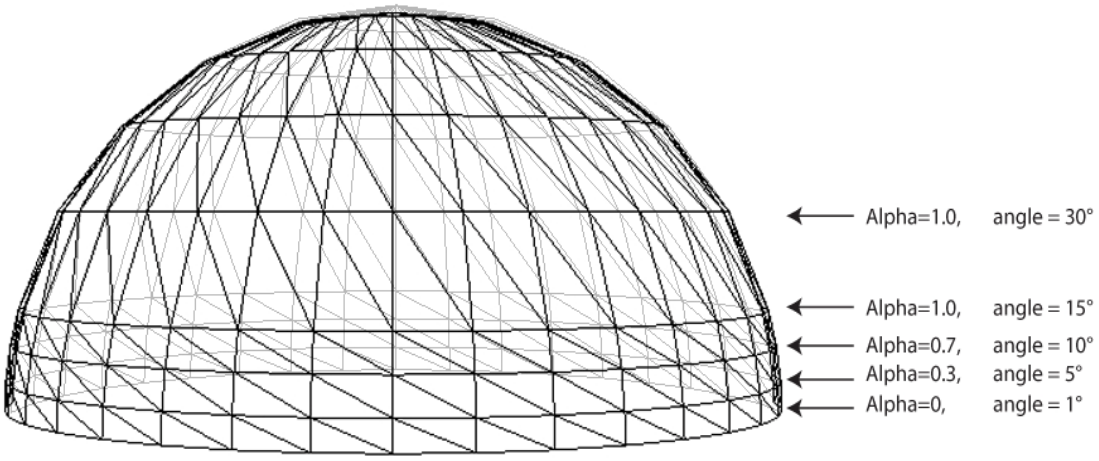

Figure 9. Cloud Dome

The dome radius is variable and must be smaller than the radius of the *SkyDome* that renders the rest of the sky. Again, the dome must be "fixed" above the viewer and this is achieved by using the *MoveEarthySkyWithEyePointTransform* transform node, which automatically translates the dome to the viewer point coordinates and the color binding is set to BIND_PER_VERTEX

c. Noise Texture

After creating the dome, a volume noise texture that will be applied onto it must be created. There are two ways to acquire the texture: either by loading the texture off the hard disk or generating it at runtime. If the supplied filename parameter is not correct or the file cannot be found, the texture is again generated at runtime. The image file format is DDS and has only pixel information in the ALPHA channel to keep the size relatively small. If the volume texture has to be generated, the *NoiseGenerator* class is instantiated and the parameters for the texture generation are: six octaves of noise, initial frequency 6, initial amplitude 0.7 and persistence 0.5. A warning message is displayed in the console to state that the supplied file name could not be loaded.

The default resolution of the volume texture is 128^3, which is a reasonable compromise between graphics memory requirements and visual appearance. Most new graphics cards have a maximum volume texture resolution of 512^3, but this would be filling up valuable space on the graphics card memory and would take too much time to be generated at runtime.

Currently, the DDS file format does not support volume texture mipmaps and, therefore, the texture minification and magnification filtering for the 3D textures is set to LINEAR. In addition, the utility OpenGL function that auto-generates mipmaps for volume textures (*gluBuild3DMipMapLevels*), either was not implemented or fully supported by all OpenGL drivers. It was, consequently, not used.

d. Shaders

Apart from the system graphics card capabilities, some ARB extensions must be present in order for GLSL to be supported. These are *GL_ARB_fragment_shader*, *GL_ARB_vertex_shader*, *GL_ARB_shader_objects* and *GL_ARB_shading_language100.* GLSL is currently supported in OSG through its core component *osgGL2*. In order to activate the shaders, which will render the clouds on the dome, some steps must be followed. GLSL introduces *vertex* and *fragment shader objects*, which contain the data structures necessary for storing the OpenGL shader. Shader objects must attach to a *program object*, which is an OpenGL-managed data structure and acts as a container for them. The shader source files must loaded into the

vertex and fragment shader objects, compiled by the OpenGL driver and linked. Then, the executable machine code is installed on the vertex and fragment processor where they will be used to render all subsequent primitives (Figure 10).

There are two shader source files: vertex source file"*cloud1.vert*" and fragment source "*cloud1.frag*". The shader source code could have been embedded in the *CloudDome* class, but having it as two separate shader files enables debugging and future modification without recompiling the application source file.

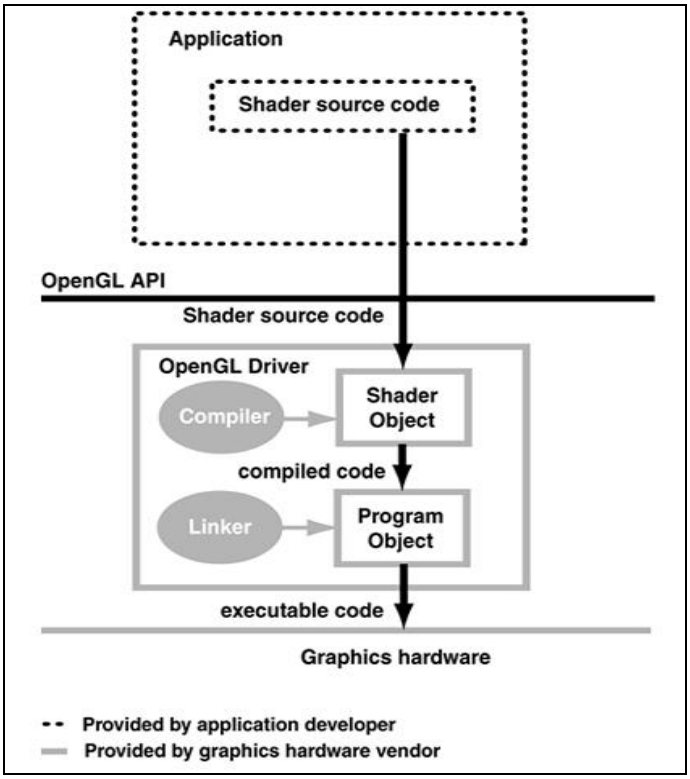

Figure 10. Execution Model for OpenGL Shaders (From: OpenGL Shading Language Book)

Communication between the application and either the vertex and fragment processor is performed by passing *uniform* variables to the vertex and fragment shaders. Uniform variables do not change across the primitive being processed and they are used as a link between the shader, OpenGL and the application. User-defined uniform variables are used by the application to pass arbitrary data values to the shaders, which allows for shader parameterization and provides greater control to the end-user. The most

important property of using uniform variables is that by modifying them at run-time, either programmatically or through a graphical user interface, realizes a variety of effects from only one shader. The cloud vertex and fragment shaders are using uniform variables to control object scale, cloud color, texture offset, color bias and other parameters. With this parameterized design, the end-user can easily visualize many different cloud textures with run-time parameter modification.

The cloud vertex shader is simple enough to create as it has only four functions to perform:

- Accepts the position of vertices in object space
- Scales the object according to a uniform variable *Scale*
- Transforms the position from object space to clip space and stores it to the *gl_position* built-in varying variable
- Passes the user-defined varying variable *ModelPosition* to the fragment shader

```
//
// Cloud Vertex Shader
// Author: George Tarantilis
//

varying vec3  ModelPosition; // Used for scaling the model

uniform float Scale; // Scale in relationship to noise texure

void main(void)
{
    // The varying variable scaled by uniform variable Scale
    ModelPosition = vec3 (gl_Vertex) * Scale;

    //Front color assignment
    gl_FrontColor = gl_Color;

    // Transform vertex position according to fixed pipeline
    gl_Position = ftransform();
}
```

Figure 11. The Cloud Vertex Shader Source File

The *Scale* uniform parameter is used originally for optimally scaling the object in relationship to the size of the noise texture, but also allows for finer control of the size of clouds. The *ModelPosition* varying variable is used for making the scaled

incoming vertex value available to the fragment shader. This would be the modeling coordinate of the object at every fragment and it is used as the input for the 3D noise texture lookup. The vertex transformation is achieved by using the built-in GLSL function *ftransform*, which "ensures that the incoming vertex position is transformed in a way that produces exactly the same result as would be produced if OpenGL's fixed functionality transform". [1]

The cloud fragment shader is where the noise volume texture is used to render clouds on the dome. It does the following:

- Accepts the user-defined varying variable *ModelPosition* from the vertex shader
- Makes the 3D texture lookup for the alpha value using the scaled model position as texture coordinates in addition to the *Offset* variable
- Performs the noise clamping and exponentiation using the *Cutoff* and *Exponent* variables, which is discussed in more detail in the "Post-Process Noise" section
- Biases the noise alpha value
- Computes the final fragment, color mixing the *CloudColor* color variable, noise alpha and vertex alpha

```
//
// Cloud Fragment Shader
// Author: George Tarantilis
//

varying vec3  ModelPosition; // Model position from vertex shader

uniform sampler3D Noise;     // The 3D noise texture
uniform vec3  Offset;        // Offset used for dynamic changing
                             // of texture coordinates
uniform float Cutoff;        // The threshold (%) of cloud coverage
uniform float Exponent;      // Affects dynamic range of noise
uniform vec3  CloudColor;    // The cloud color
uniform float Bias;          // Bias variable used for adjusting
                             // the final color

void main (void)
{
    // Make a lookup of the texel alpha value of the noise texture
    // according to ModelPosition and the Offset uniform variable
    float  noise  = (texture3D(Noise, ModelPosition + Offset)).a;

    // Clamping of noise values
    // Noise changes 0..1 -> 0..1-Cutoff
    if(noise < Cutoff)
        noise = 0.0;
    else
        noise -= Cutoff;

    // Exponentiation of the noise and biasing to give finer control
    noise = Bias * (1.0 - pow(Exponent, noise));

    // Combine computed alpha value with CloudColor uniform variable
    // and push the fragment down the pipeline for the blending
    // operation with the background sky
    gl_FragColor = vec4(CloudColor, noise * gl_Color.a);
}
```

Figure 12. The Cloud Fragment Shader Source File

For each incoming fragment, the cloud fragment shader computes its color by writing this value into the special output variable gl_*FragColor*. For determining the final fragment color, the red, green and blue channel information of the *CloudColor* color variable is taken and combined with the vertex alpha value, which has been transmitted internally from the vertex shader and modulated by the already computed *noise alpha* variable. The results of the fragment shader are then sent on for further processing.

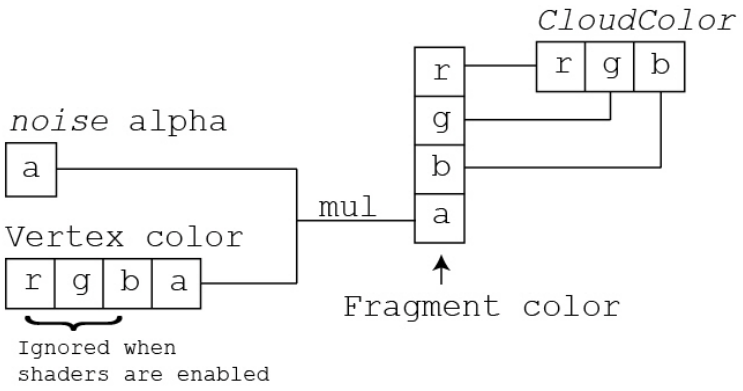

Figure 13. Fragment Color Computation

The remainder of the OpenGL pipeline remains the same. Fragments are submitted to coverage application, pixel ownership testing, scissor testing, alpha testing, stencil testing, depth testing, blending, dithering, logical operations, and masking before ultimately being written into the frame buffer. The only operation used beyond the fragment processor in this application (*CloudDome::Create* function) is blending. The *gl_FragColor* final cloud color value must be blended with the sky. The source and destination factors used are SRC_ALPHA and ONE_MINUS_SRC_ALPHA, respectively.

To simulate procedural clouds that are forming, disappearing and drifting across the sky, instead of using only the scaled dome position as the index (texture coordinates) into the 3D noise texture, an offset value is added. This offset is defined as a uniform variable (*Offset*) and can be updated by the application each frame. The clouds can drift slowly by modifying the x component of this uniform variable while the cloud forming is controlled with the y component. To achieve a more complex effect, both coordinates can be modified each frame.

e. Update

All user-defined uniform variables are initialized in the application and updated with the second method described in the *CloudPlane* class. *CloudDome* class registers to receive messages from the *dtCore::System* class with the *CloudDome::OnMessage( )* method. By altering the values of *Exponent* and *Cutoff*

variables, the user can control the clamping and exponentiation process in real-time. Similar control is attainable with *CloudColor*, *Bias* and *Offset* uniform variables. All these variables are defined as private data members in the class but they have their *get( )* and *set( )* methods that allow access and modification.

C. SUPPLEMENTARY RESEARCH

1. Post-Process of Noise Textures

2D noise textures generated with *NoiseGenerator* must be clamped and exponentiated. This parameterized process will produce noise textures with a cloud-like appearance. The pixel depth of the texture is eight bits because it only has the alpha channel available. Originally, these alpha values vary between 0 and 255 (or 0 to 1 in OpenGL format), which are all possible values of a 1-byte data type. The user chooses the *cutoff* and *density* values that will be used in the clamping and exponentiation process.

The two functions can be combined in a two-leg function. If *alpha* is the original alpha value of a pixel in the texture, it is modified as follows:

- If alpha < cutoff → alpha = 0
- If alpha > cutoff → alpha = $1 - \text{density}^{\text{alpha-cutoff}}$

This modification has the effect of shrinking the dynamic range of the texture and altering the transitions at the cloud edges. Figure 14 shows some sample curves representing the output alpha values when processed with some typical values of *density* and *cutoff*.

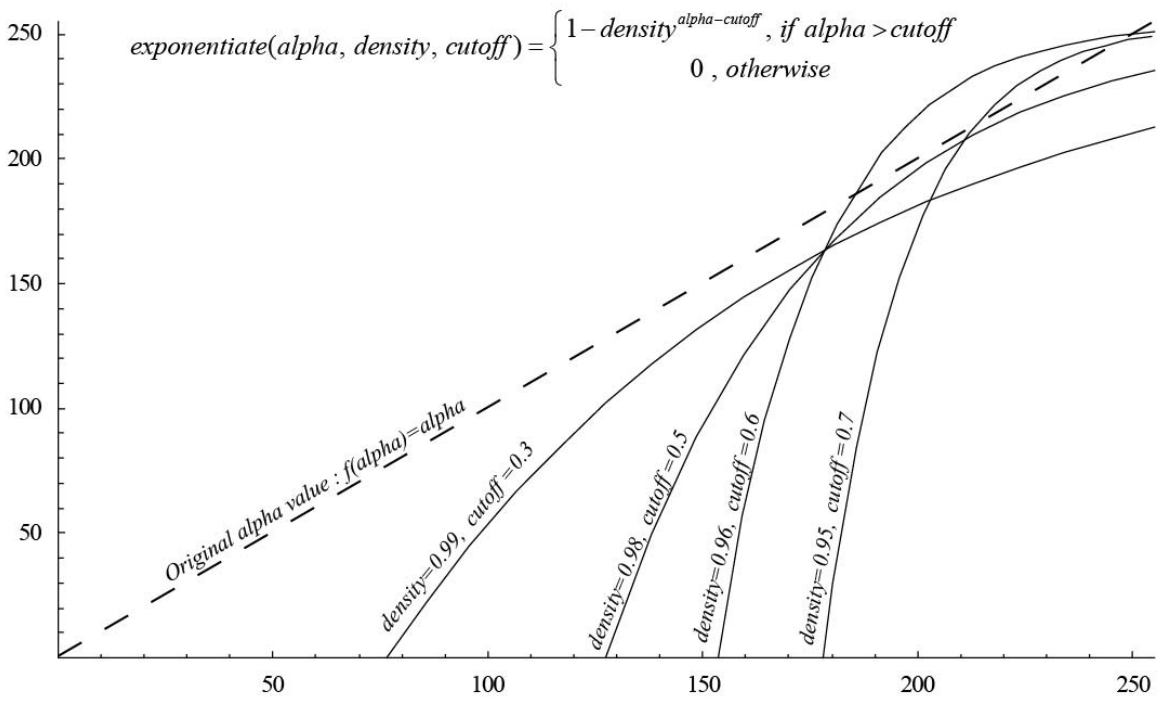

Figure 14. Noise Clamping and Exponentiation Equation

The curve with density equal to 0.99 and cutoff equal to 0.3 will give the effect of a near-filled sky with smoothed cloud edges and a very narrow dynamic range, while the one with density equal to 0.95 and cutoff equal to 0.7 will generate a texture with sparse clouds and hard edges.

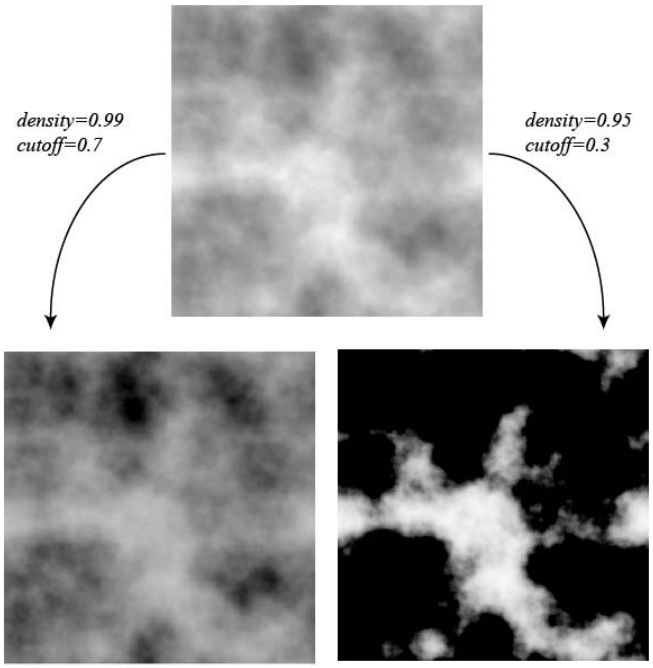

Figure 15. Exponentiation Effects on Noise Texture

It is obvious that with various combinations of these two parameters along with the modification of the original parameters of the noise function (octaves, initial frequency, initial amplitude and persistence), a nearly endless variety of cloud-like textures can be produced. These textures were generated in volatile memory and they had to be stored to some permanent storage media, such as a hard disk, using an appropriate image file format. The problem faced was that not many image file formats are available that support volume textures.

2. DDS File Format

The generated 2D and volume noise textures were stored as DDS (DirectDraw Surface) files. This file format is created by Microsoft and is used to store 2D and volume textures and cubic environment maps, both with and without mipmap levels. This format can also store uncompressed and compressed pixel formats, and is the preferred file format for storing DXT compressed data. This file format is supported by Microsoft with the *DirectX Texture tool* (DXTex Tool) and also by NVIDIA, which has provided several useful tools such as *nvdext*, a *DDS import plugin* for Photoshop and a *DDS thumbnail viewer* for Windows 2000/XP.

The internal format in which OSG stores images (*osg::Image*) is a thin encapsulation of the OpenGL image data structure. The way OSG supports different image *file* formats for importing and exporting images is via the *Reader-Writer* plugin mechanism. Plugins allow users to write code to read arbitrary file types and convert them into native formats. The plugin architecture also allows users to create a writer mechanism that converts internal format images into arbitrary files types. As of May 2004, some formats were available for exporting 2D images (**.rgb, *.sgi, *.bmp, *.jpg* and **.pnm)*, but most had only *Readers* and none could export volume textures. This limitation was important because it was not possible to store the *NoiseGenerator* volume noise textures to hard disk, which led to the development of the missing necessary support in OSG's DDS plugin: volume image file import and export and 2D image file export.

Figure 16 shows the DDS file layout. Its parts are:

- A value at the file header used to identify the file as the DDS format
- A "Surface Format Header" that contains all the information needed to determine the contents of the entire file.
- A "Main Surface Data" area that contains the actual image data (pixel values according to the pixel format: RGB, ALPHA, LUMINOSITY, etc.)
- An "Attached Surfaces Data" area that stores additional image data for mipmaps or cubemaps

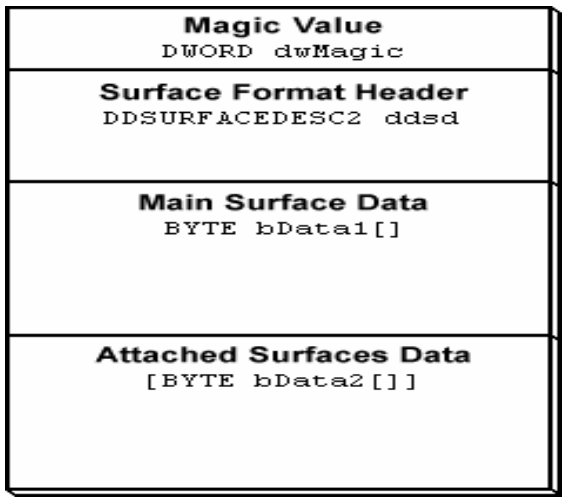

Figure 16. Structure of the DDS Image File Format

The DDSURFACEDESC2 structure contained in the surface format header describes the file contents using the standard flags and values defined in the Microsoft DirectDraw documentation. For example, for a volume texture to be written to disk, among other things, the following DDSURFACEDESC2 structure flags must be set:

- ddsd.dwFlags |= DDSD_DEPTH
- ddsCaps.dwCaps |= DDSCAPS_COMPLEX
- ddsCaps.dwCaps2 |= DDSCAPS2_VOLUME

The modifications to the DDS plugin were accepted and included in the OSG open-source repository.

3. "Make Some Noise" Tool

One problem encountered was how to find the correct values that would create the textures envisioned. Figure 17 shows the initial approach.

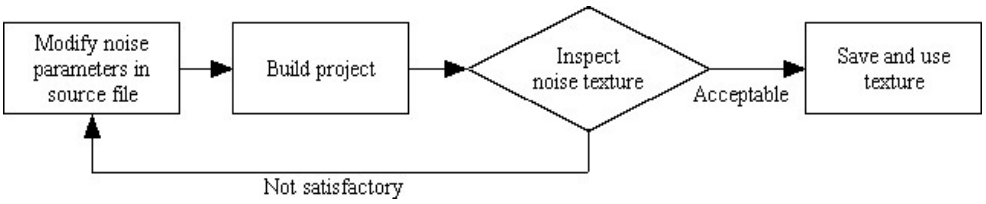

Figure 17. Noise Texture Creation Process

This highly iterative and time-consuming process had to be more efficient. A GUI tool was created with Troll Tech's *Qt* v.3.2.1(Non-commercial version) that helped to streamline the process and freely tweak any parameter obtaining immediate visual results. Also, the "Save" function to 2D or 3D DDS files was a method to store interesting textures to the hard disk and use them in the *CloudPlane* and *CloudDome* classes.

A modified version of *NoiseGenerator* class was used with some additional functionality:

- Option to generate textures as RAW image data instead of only *osg::Image* textures
- Included *exponentiation* function

These changes rendered the *NoiseGenerator* class OSG agnostic and suitable for a broader range of applications. Additionally, the following features justified the choice of Qt as the GUI builder instead of other toolkits:

- Excellent portability of the code in Linux and Macintosh platforms
- Complete help system
- Tight integration with Microsoft Visual Studio
- Specialized tools for rapid code development

Figures 18 and 19 show the tool ported to the Mac OSX (Panther) and Fedora Core 2 Linux distribution, respectively.

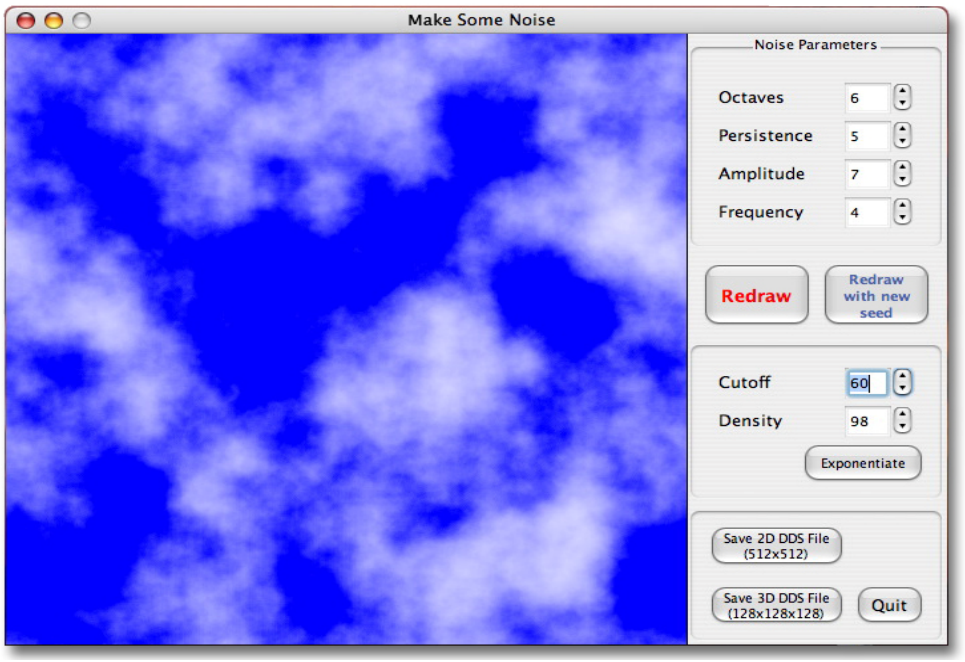

Figure 18. "Make Some Noise" Tool Ported to Mac OSX

With this tool, the process of choosing the correct values for the noise parameters was now straightforward. Ample textures, which represent various conditions of cloud coverage, density, and texture, can be generated and stored for later use.

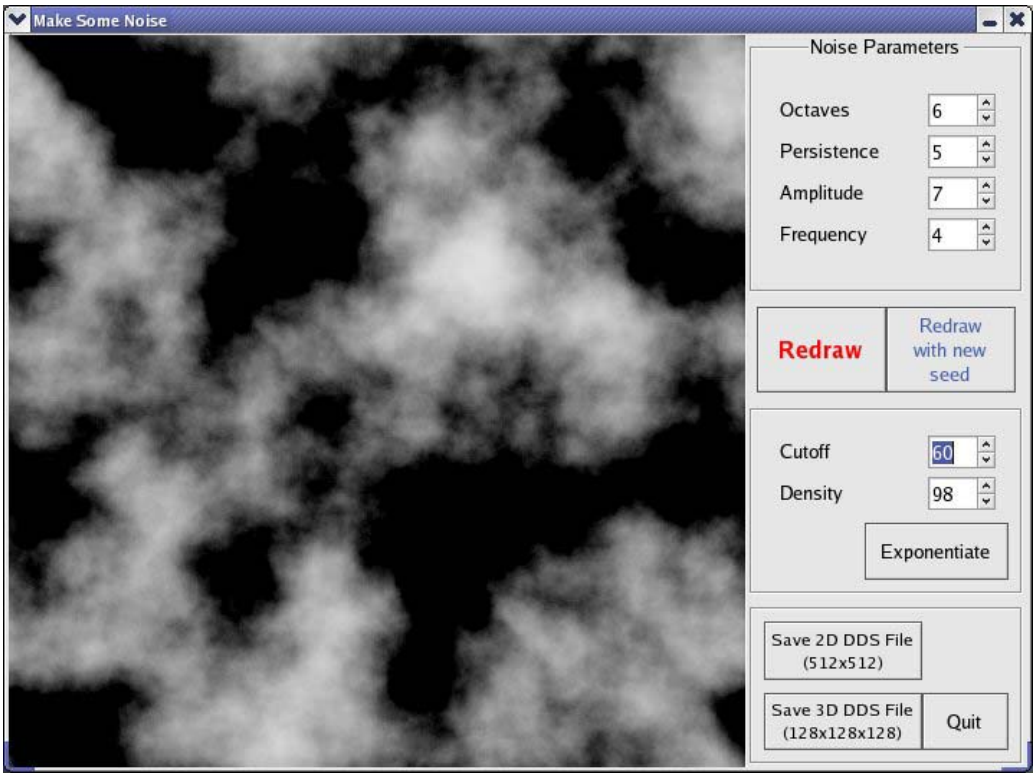

Figure 19. "Make Some Noise" Tool Ported to Linux (Fedora)

4. Shader Development

The development of shaders in a high-level shading language such as GLSL or Cg require many trials and iterations because these languages are still new and their features, power and weaknesses are not yet fully understood. Additionally, programming the graphics hardware requires precise knowledge of the way it replaces the fixed pipeline, what actions are allowed at every stage and which underlying hardware will be targeted. Moreover, during the development of shaders, there is no feedback where logic errors exist and the debugging is very limited.

These difficulties can be addressed by using new tools created for shaders development. One is ATI's Shader IDE RenderMonkey in which the cloud vertex and fragment shaders were created. When first introduced, it was available only for the HLSL and Cg languages, but the 1.5 version supports GLSL as well. This application is freely distributed and simplifies shader creation by providing an integrated environment for editing and compiling shaders, loading textures, creating varying and uniform variables and providing instant visual feedback. Also, it is an excellent tool for prototyping and

debugging new graphics algorithms. In addition, it is possible to define multiple passes for the fragment shaders if the underlying graphics hardware cannot support loops or control structures in them.

Figure 20 is a screenshot of the RenderMonkey application. All windows can be customized and positioned at will. Typically, on the left side, there is a tree structure of the workspace where the user can define uniform variables, textures, render states, camera objects and models to apply the shaders. In the "Artist Editor" window, there are sliders by which the user can modify the uniform variables. In the *Output* window, messages can be viewed from the compiling and linking process, along with other run-time messages. The vertex and fragment shader source files can be edited with syntax-highlighted formatting in the *Shader Editor* window. A *Preview* window is available to monitor the effects that shaders have in the loaded 3d models.

Shader development time is reduced dramatically using this IDE as workspaces can be templated, saved and reused. Using the GLSL shaders in the OpenGL application is straightforward. The user simply needs to set the render states and define the textures and uniform variables.

Other available non-commercial IDEs for shaders are NVIDIA's *FX Composer*, which support Cg and HLSL, and TyphoonLabs' *Shader Designer*, which supports GLSL.

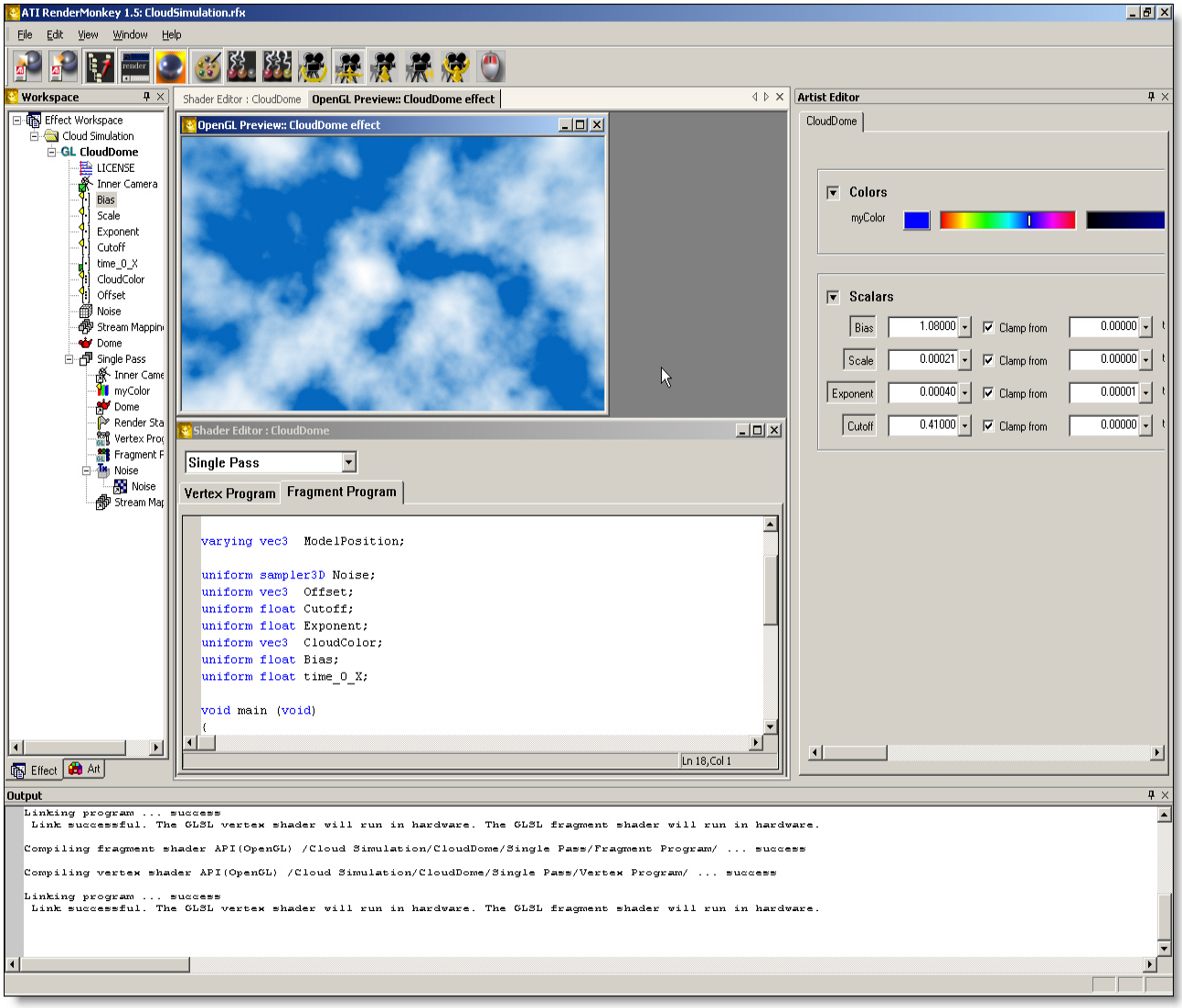

Figure 20. ATI's RenderMonkey IDE

THIS PAGE INTENTIONALLY LEFT BLANK

IV. CLOUD SIMULATION IMPLEMENTATION

A. DELTA3D INTEGRATIONS

The *NoiseGenerator, CloudPlane* and *CloudDome* classes have been integrated in the Delta3D simulation engine and can be used by any application needs more realism in outdoor scenes. A simple test application has been created to exhibit the capabilities of these classes and serve as one of the demonstrations for the MOVES Open House 2004. Performance measurements have been conducted with the *Fraps* benchmark tool to evaluate the performance penalty that cloud rendering imposes on the test application.

1. CloudPlane

Creating layers of clouds inside a simulation with the *CloudPlane* class is very simple. One or more layers can be rendered with the constraint that the order of the layers is from the furthest to the closest due to OpenGL transparency and blend function management. The declaration of a layer is as follows:

```
dtCore::CloudPlane *cloudPlane;
```

or if OSG referenced pointers is used

```
osg::ref_ptr<dtCore::CloudPlane> cloudPlane;
```

Each layer must be instantiated passing parameters to the constructor that defines the characteristics of the cloud plane: noise parameters (octaves, cutoff, frequency, amplitude, persistence, density and texture size), height from the ground and optionally name.

```
cloudPlane = new dtCore::CloudPlane(octaves, cutoff, frequency,
amplitude, persistence, density, texSize, height, "Low layer");
```

Then each cloud layer must be added to the *Environment,* as all *EnvEffects* are required in order to be populated in a list and rendered correctly. A handle to a *Weather* instance is necessary since the *Environment* instance is, by aggregation, included in *Weather*:

```
weather->GetEnvironment()->AddEffect(cloudPlane);
```

Multiple layers can be added to the *Environment* with the same procedure, which in turn must be added as a *Drawable* to the main *dtABC::Application*:

```
application->AddDrawable(weather->GetEnvironment());
```

In the sample test application *testClouds*, an array of three elements was initialized with the following three cloud layers of different characteristics:

```
osg::ref_ptr<dtCore::CloudPlane> cloudPlane[3];
// Overcast sky with clouds
cloudPlane[0]=new dtCore::CloudPlane(10, .6, 3, .5, .7, .95, 1024, 1400);
// Broken clouds
cloudPlane[1]=new dtCore::CloudPlane(10, .7, 6,  1, .4, .96, 1024, 1200);
// Few Clouds
cloudPlane[2]=new dtCore::CloudPlane(8,  .8, 16, .8, .4, .96, 1024, 900);
```

These are added from the furthest (1400 units) to the closest (900 units) with decreasing cloud covering area (*Cutoff* increasing from 0.6 to 0.8) and increasing frequency to render small clouds at the bottom layer. The textures of the layers could be any power of 2 resolution depending on the application. A typical resolution is 512x512 pixels, while 1024x1024 could be used for smoother textures and 256x256 when there are memory constraints. These three 1024x1024 layers require 3 Mb of graphics memory, which is within limits of most graphics cards. If all four channels (RGBA) had been used for the textures, the required memory would have been 12 Mb, so the choice of using only an alpha channel texture saves 8 Mb of memory in this particular occasion. Generally, the memory requirements for only alpha channel textures are ¼ of the regular RGBA textures.

The translation of the texture coordinates of the quads relative to their height from the x-y plane, along with the addition of a terrain model, augments depth perception of the scene. The terrain is constructed by means of the *dtCore::InfiniteTerrain* class, and this scene was the “test bed” on which the cloud classes were tested. The following screenshots in Figures 21, 22 and 23 represent each *CloudPlane* layer, with the parameters mentioned earlier: “Overcast sky”, “Broken clouds”, “Few Clouds”. Layers can be combined and rendered one on top of the other as shown in Figure 24 where layers 2 and 3 have been combined.

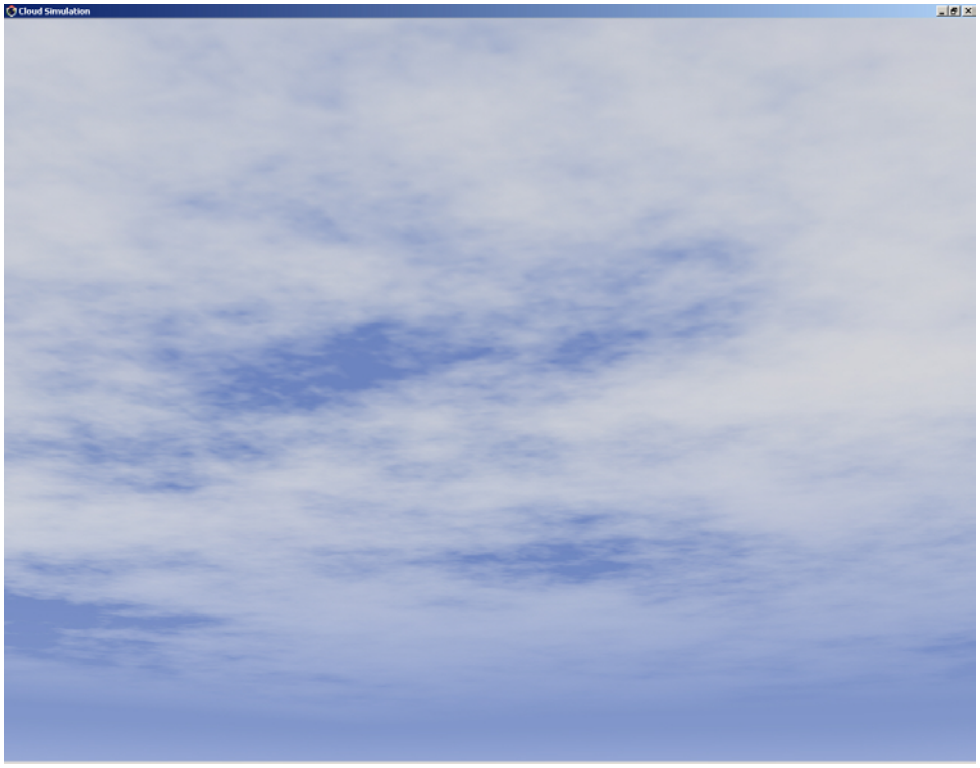

Figure 21. Overcast Sky

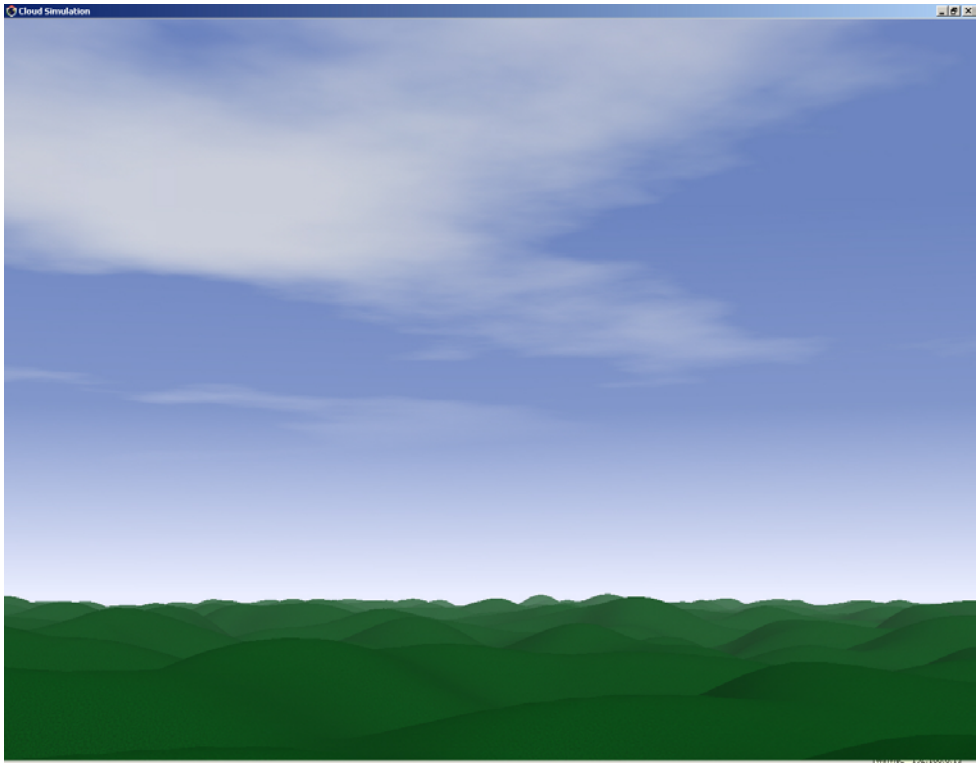

Figure 22. Broken Clouds

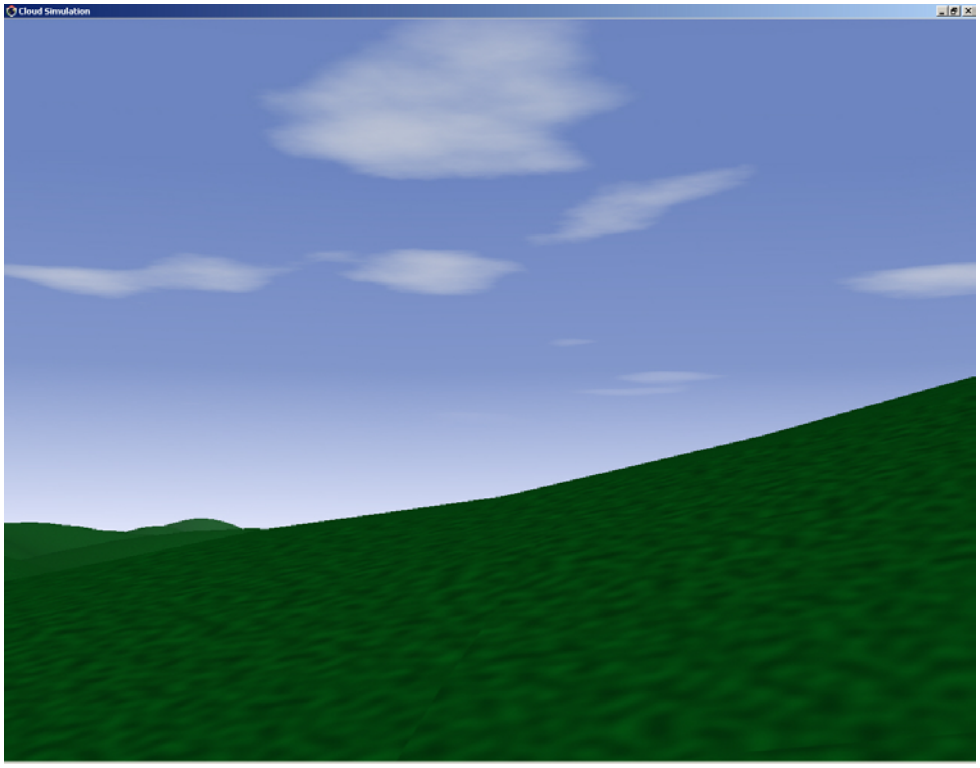

Figure 23. Few Clouds

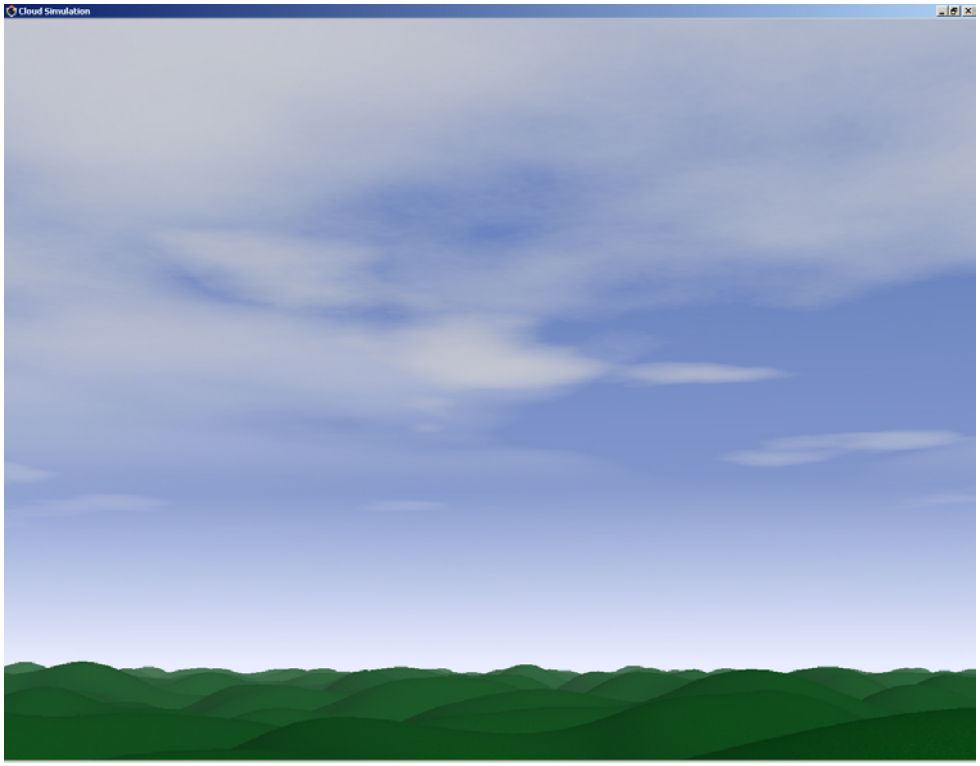

Figure 24. Combined 2nd and 3rd Layers

The cloud layers are shaded according to the time of day and appear different shades according to their height from the x-y plane. Due to the loose tessellation of the cloud quads, the color is uniform across the inner vertices. This feature, in addition to the built-in color model of the sky, adds realism and visual appeal to the simulation. If fog has been added to the scene, setting the respective OpenGL fog state fogs the cloud layers as well individually. Figure 25 shows a scene at dusk:

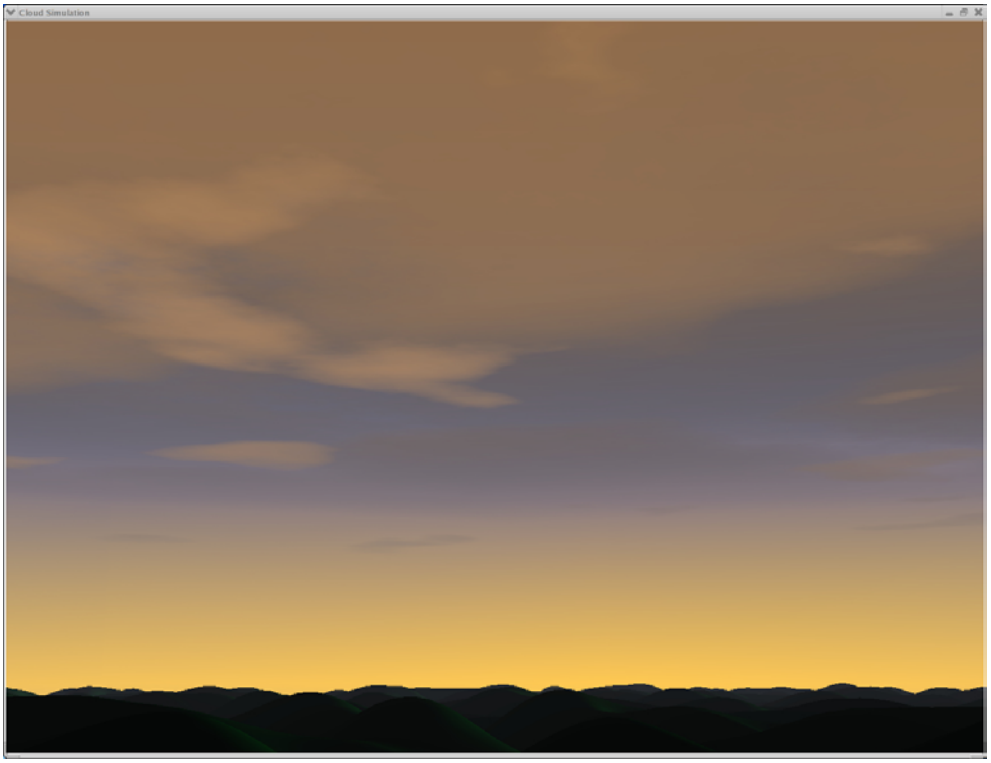

Figure 25. Clouds with CloudPlanes at Dusk

2. CloudDome

In order for a system to use OpenGL shaders, it must fulfill two prerequisites: a capable graphics card and OpenGL drivers that support GLSL. The declaration of the second cloud rendering class, *CloudDome*, is no different from the *CloudPlane*:

```
dtCore::CloudDome *cloudDome;
```
or if OSG referenced pointer is used
```
osg::ref_ptr<dtCore::CloudDome> cloudDome;
```

There are two ways to define the 3D noise texture used by the *CloudDome* class: either by directly loading a pre-computed 3D noise texture or by computing one at runtime. The same noise parameters (octaves, cutoff, frequency, amplitude, persistence and density) as in the *CloudPlane* class are required, except from the texture size, which is fixed to 128x128x128. Additionally, two extra parameters should be provided: one that represents the radius of the dome and one for the segments of the circumference of the dome. Consequently, there are two constructors that can handle these two cases.

```
cloudDome = new dtCore::CloudDome(octaves, cutoff, frequency,
amplitude, persistence, density, radius, segments);
```

The second constructor only requires these two parameters and the DDS filename of the pre computed 3D noise texture:

```
cloudDome = new dtCore::CloudDome(radius, segments, <filename>);
```

These two constructors can be used interchangeably but only one is allowed in any application. The second form is faster in loading time but it requires a texture. An example of a *CloudDome* instantiation can be found in the "Cloud Simulation" demo in which the 3D texture is computed on the fly. The following constructor has been used:

```
cloudDome = new dtCore::CloudDome(6, 2, .7, .5, .7, 5, 5500, 20);
```

The *CloudDome* instance must be added to the *Environment* in order to be included in the *EnvEffect* list and rendered correctly.

```
weather->GetEnvironment()->AddEffect(cloudDome.get());
```

Again, the *Environment* must be added as a *Drawable* to the main *dtABC::Application*:

```
application->AddDrawable(weather->GetEnvironment());
```

The values of all the uniform variables of the shaders can be conveniently modified after an *CloudDome* object is constructed by using their respective setter

functions: *setScale*( ), *setExponent*( ), *setCutoff*( ), *setSpeedX*( ), *setSpeedY*( ), *setBias*( ), *setCloudColor*( ). Changing these variables has the effect of real-time alteration of the cloud appearance. Furthermore, combining values into themes can be helpful for storing certain cloud appearances for later use.

For the needs of the "Cloud Simulation", the FLTK applet, which was described in a previous section, was used to modify these values by user-interaction and non-programmatically.

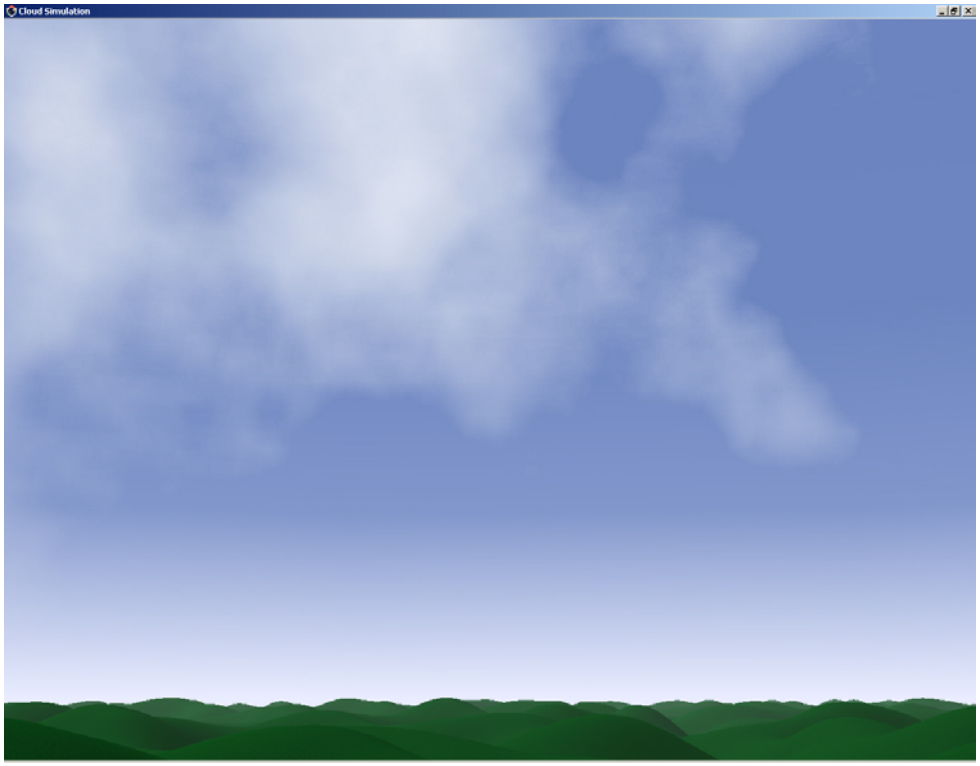

Figure 26. CloudDome Screenshot 1

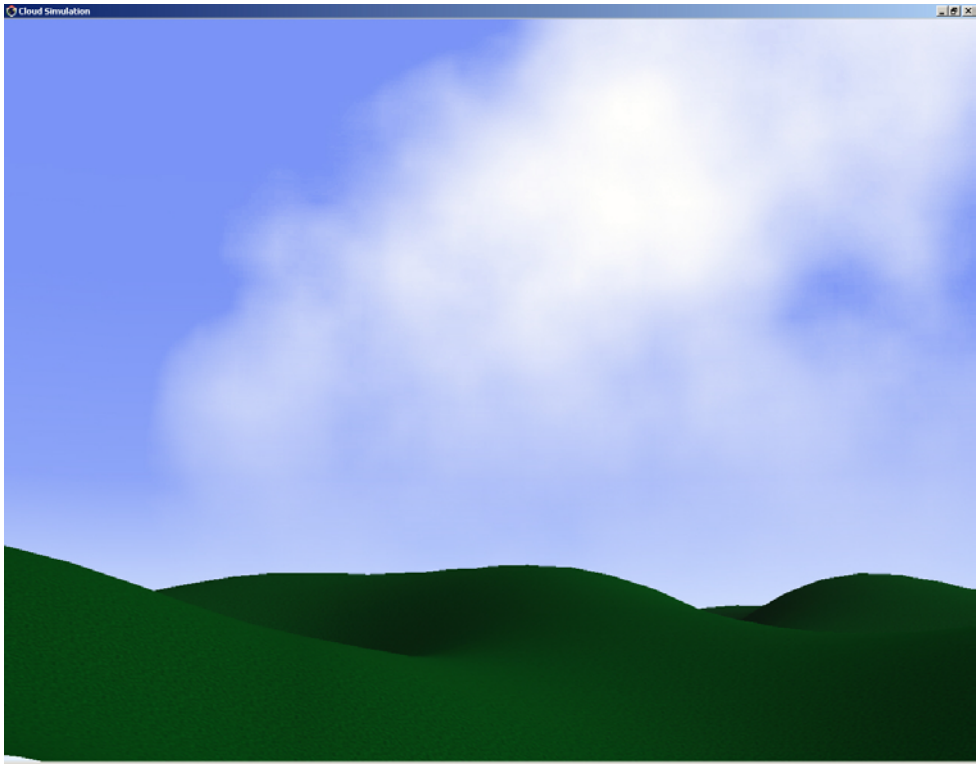

Figure 27. CloudDome Screenshot 2

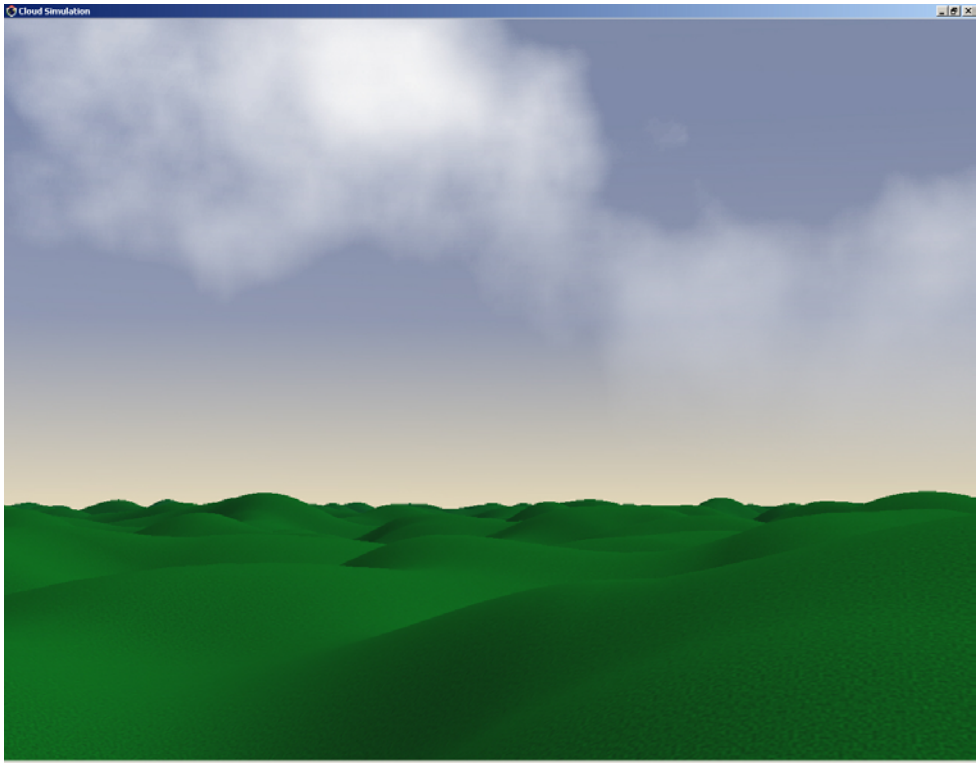

Figure 28. CloudDome Screenshot 3

3. FLTK GUI

The *CloudDome* shaders can be previewed with the use of a GUI applet in which the user has access to all parameters (uniform variables) of the shaders. This applet is created with a FLTK User Interface Designer (*fluid*), which Delta3D had already been using as a GUI for testing other types of classes. The *Offset* variable is replaced by *X speed* and *Y speed* variables, which denote the horizontal and vertical cloud speed. The user is given the capability to change every aspect of the clouds appearance effortlessly and instantly observe the results. Also, the applet provides the option to disable the shaders completely and enable back the OpenGL fixed functionality for debugging, performance benchmarking or other reasons.

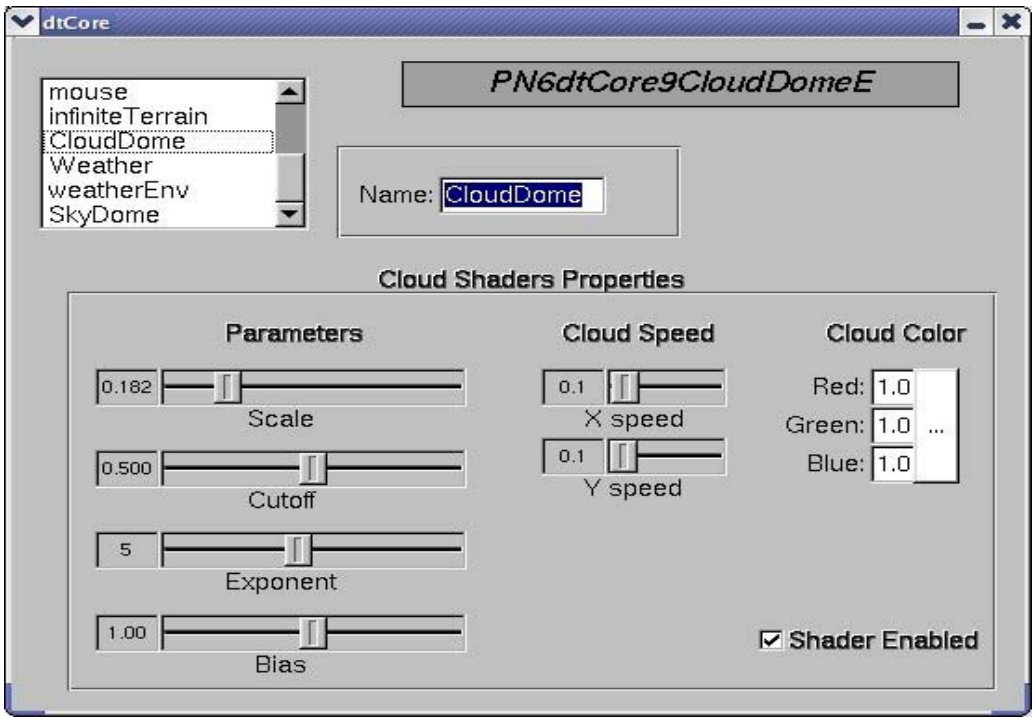

Figure 29. FLTK GUI Applet for CloudDome

For example, the screenshots in Figures 29 and 30 are from the same static scene and show the "forming" of clouds by simply altering the uniform parameter *Cutoff* via the FLTK GUI.

Figure 30. Cutoff Equal to 0.85 and 0.78

Figure 31. Cutoff Equal to 0.72 and 0.61

4. Performance

In order to evaluate the performance of *CloudPlane* and *CloudDome* classes, the frames per second of the "Cloud Simulation" have been measured before and after the use of any cloud rendering method. All the measurements have been conducted on a system with the following characteristics:

- Processor: Intel Pentium 4 HT, 2.933 Mhz
- Motherboard: MSI 865PE, 1Gb RAM
- Video Adapter: ATI All-in-Wonder 9600 (128 Mb), Catalyst drivers v. 4.7
- Display Mode: 1280x1024 with 32 bits per pixel, Refresh rate: 100Hz
- Operating System: Microsoft Windows XP with SP1

The tool used for the measurements was *Fraps* v.2.0. The Vsync property was disabled in order to avoid artificial capping of the FPS due to the graphics card and monitor synchronization. This decision was taken because Delta3D itself does not have

an internal limit that would have prevent applications from rendering faster than a certain frame rate. In other words, the maximum frame rate that the application can obtain will not be limited by the refresh rate of the monitor. During the measurements, all unnecessary processes and applications were shutdown and the simulation executed in full screen. Finally, the Windows XP internal limit for OpenGL applications (60 FPS) was disabled.

The results, which are presented in Table 1, show that overall, there is a drop in the frame rate as expected, but this drop is not always significant. Specifically, the frame rate when one *CloudPlane* was activated had a drop of 15%. The maximum frame drop occurred when *CloudDome* was activated where a drop of 47% made the actual FPS equal to 44. Even then, it was still greater than the minimum 30 FPS required for correct animation perception.

These findings prove that that the addition of cloud rendering techniques has some adverse performance effects on the application, however, these natural effects can be rendered at real-time in mainstream graphics cards.

Sky State	Frames Per Second	Drop (%)
Empty sky	83	-
One CloudPlane	70	15
Two CloudPlanes	62	25
Three CloudPlanes	59	29
CloudDome	44	47

Table 1. Application FPS Changes with the Addition of Clouds

B. LINUX PORT

At the time of this writing, the Delta3D simulation engine had been developed only for Microsoft Windows platforms, but the goal is to be available for other platforms as well. As far as the Linux platform is concerned, plenty of non-commercial Linux distributions are suitable for developing any kind of large-scale applications including virtual environment simulations. Both major graphics cards vendors (ATI & NVIDIA)

are maintaining up-to-date Linux OpenGL drivers along with support for OpenGL shaders. In addition, the main underlying framework on which Delta3D is based, OSG, can be natively built on Linux machines.

Most Delta3D simulation engine components have been ported to Fedora Core 2 Linux distribution as the final portion of this thesis. In order for the "Cloud Simulation" to be able to execute correctly in this platform, some modifications in the source code were necessary. The most important modifications were:

- Several class name conflicts had to be resolved
- Strict filename and code case (uppercase and lowercase letters) had to be enforced
- Some alternate standard C++ library function definitions had been provided
- Recompilation of all of the dependencies of Delta3D under Linux (FLTK, pLib, InterSense, tinyXML, ReplicantBody, Cal3D, ODE and OpenThreads, Producer and OpenSceneGraph)

The resulting code had similar or better performance compared to that of the Windows platforms. The actual MOVES Open House "Cloud Simulation" application was demonstrated on a Linux machine.

V. CONCLUSIONS – FUTURE WORK

A. SUMMARY

This thesis designed and implemented a simulation component that renders clouds using procedural noise-based texturing techniques. Two modeling approaches were used. In the first method, procedural textures representing cloud layers were mapped onto elevated quads that were properly tessellated. The second method employed OpenGL Shading Language to update the position and other characteristics of a 3D procedural texture mapped onto a dome dynamically. Both methods create convincing clouds when used in interactive outdoor simulations while the performance penalty imposed is considered acceptable.

The 2D and 3D noise textures used in both methods were created having a seamless tiling property so that their edges would not be noticeable. Additionally, 3D texture read/write support was built for the DDS file format plugin of OpenSceneGraph in order to store the textures for later use. A custom GUI tool, "Make Some Noise", has also been built to help with the efficient creation and storing of 2D and 3D textures through parameterization.

In addition, cross-platform compatibility was demonstrated by porting the *Cloud Simulation* to Fedora Linux. Lastly, the cloud simulation component was included in the Delta3D simulation engine and was used in demonstrations during MOVES 2004 Open House.

B. FUTURE WORK

Future work should focus on expanding the capabilities of the current cloud simulation component concerning the visual detail and quality and volumetric support but also on developing support for the simulation of other natural phenomena such as rain, haze, snow, and lighting.

1. Clouds

The visual detail of cloud simulation could be improved by developing a more sophisticated shader algorithm that could not only modify texture coordinates and texture exponentiation, but also create turbulent flows and swirls in the clouds. In addition, one

restriction of the current cloud simulation component is that it limits the viewpoint around the ground level. This limitation could be removed if the volumetric cloud modeling approach was followed. This approach is recommended for simulations that demand flight in and around clouds.

Presently, the shading of clouds allow only for one color due to the nature of the texture mapping technique used. A new shading model could be developed for greater realism of the cloud simulation during dawn and dusk hours that could give clouds self-shadows and color variation.

2. Other Natural Phenomena

The inclusion of models of various other natural phenomena such as rain, snow, and haze in outdoor simulations greatly enhances realism and fidelity. Especially in training applications, the representation of various weather conditions is often necessary. A library of such phenomena could be built in the Delta3D simulation engine using shaders or traditional rendering techniques. For example, rain could be modeled as antialiased, blurred lines rendered on a screen-aligned rectangle. The lines could be slanted according to the viewpoint speed.

The possibilities are endless with the number of effects that could be modeled and included in the simulation engine. When the graphics hardware matures further, it should be possible to generate these natural phenomena on-demand at runtime without a significant drop in performance.

APPENDIX GLOSSARY

2D	Two dimensional
3D	Three dimensional
Mipmap	Texture maps of decreasing resolutions used to antialise texture map primitives
Vertex	A point in three-dimensional space
Fragment	The set of data that is generated by rasterization and represents the information necessary to update a single frame buffer location
Shader	Source code written in a shading language that is intended to executed in the vertex and fragment processors
FPS	Frames per Second
GUI	Graphical User Interface

THIS PAGE INTENTIONALLY LEFT BLANK

BIBLIOGRAPHY

All brands and names are trademarks of their respective companies.

[1] Randi J.Rost (2004). *OpenGL Shading Language*. Addison Wesley © 2004 ISBN 0-321-19789-5.

[2] David S. Ebert et al., (2002). *Texture & Modeling – A Procedural Approach (3rd Ed)*. Morgan Kaufman © 2002 ISBN1-55860-848-6.

[3] Ken Perlin (2002). *Improving Noise*. SIGGRAPH 2002 Paper.

[4] David S. Ebert et al., (2003). *A Real-Time Cloud Modeling, Rendering and Animation System*. Eurographics/SIGGRAPH Symposium on Computer Animation (2003).

[5] Yoshinori Dobashi et al., (2000). *A Simple, Efficient Method for Realistic Animation of Clouds*. SIGGRAPH 2000 Paper.

[6] Tomas Akenine-Möller and Eric Haines (2002). *Real-Time Rendering (2nd Ed)*. A K Peters © 2002 ISBN 9-781568-811826.

[7] Randima Fernardo and Mark J. Kilgard (2003). *The Cg Tutorial.* Addison Wesley © 2003 ISBN 0-321-19496-9.

[7] Matt Welsh et al., (2002). *Running Linux (4th Ed)*. O'Reilly © 2002 ISBN 0-596-00272-6.

[8] John A. Day (2003). *The Book of Clouds.* Barnes and Noble © 2003 ISBN 0-7607-3536-0.

[9] Randima Fernardo et al., (2004). *GPU Gems.* Addison Wesley © 2004 ISBN 0-321-22832-4.

[10] Mark J. Harris. *Real-Time Cloud Simulation and Rendering*. University of North Carolina Technical Report #TR03-040. 2003.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Rudy Darken
   MOVES Institute of Naval Postgraduate School
   Monterey, California

4. CDR Joseph Sullivan
   MOVES Institute of Naval Postgraduate School
   Monterey, California

5. Erik Johnson
   MOVES Institute of Naval Postgraduate School
   Monterey, California

6. LT Georgios Tarantilis
   65 Ag Spiridonos, Egaleo, 12243
   Athens, Greece